Tampere University

Ville Lepistö

USING BARCODES TO RETURN UNUSED CASH PAYMENT FROM AN OUTDOOR PAYMENT TERMINAL

Master of Science thesis
Faculty of Engineering and Natural Sciences
Examiners: Professor Matti Vilkko and Project Manager Jari Seppälä
October 2020

ABSTRACT

Ville Lepistö: Using barcodes to return unused cash payment from an outdoor payment terminal
Master of Science thesis
Tampere University
Automation Engineering
October 2020

This thesis focused on solving an issue present in a customer's outdoor fuelling terminal network. The terminals were unable to return change for interrupted cash fuellings. Manually returning the unused cash to end users used up the resources of the customer's customer service and accounting teams. Customer wanted to automate the cash return process to save resources. The focus of this thesis was on the design and implementation process of the new automated feature. This thesis aimed to answer how much resources could be saved by automating the return process, and what were the optimum technologies to implement the features within the constraints of the existing network.

The work started by researching other existing solutions and implementations on similar issues to gain a better understanding of the problem. The research initially led to three different methods of approach for solving the problem. Out of the three, an approach based on generating and redeeming voucher codes in place of cash returns was selected. After the selection, further design questions were contemplated with the customer, which lead to the determination of the use cases and requirements.

The solution was designed with the help of literature and online sources. The design finalized as a solution where a barcode containing the voucher code would be printed in the transaction's receipt in case a cash transaction was interrupted. This barcode was to be scanned at the beginning of a new transaction at any terminal in the network. From there it was to be validated against a central voucher database controlled by Voucher-application implemented as a REST API. The monetary amount stored related to the voucher code would then be added to the new transaction, returning the unused amount to the user.

The optimal implementation technologies were selected based on the conducted research. QR Code was selected as the optimal barcode symbology and GUID as the voucher code format. The Voucher-application was implemented as an ASP.NET Core Web API and it was to be hosted On-Premise in the customer's terminal network. The solution was then implemented.

By the time of writing, the new features were only enabled on two stations as pilot software. This had not produced enough data draw any unambiguous conclusions about the functionality of the new features. However, some results could be interpreted from the data. The data suggested that the system was functioning as intended and the selected technologies were suited for the purpose. During the entire pilot phase, the two stations had managed to automate approximately 4.75 hours of labour, which suggested that once widely distributed, the new features would considerably save the customer's resources.

Keywords: payment terminal, cash return, barcode, filling station, .NET, REST

TIIVISTELMÄ

Ville Lepistö: Automaattimaksun käyttämättömän käteisen palauttaminen viivakoodien avulla
Diplomityö
Tampereen yliopisto
Automaatiotekniikka
Lokakuu 2020

Tämä diplomityö keskittyi ratkaisemaan asiakkaan polttoaineautomaateissa olleen ongelman. Automaatit eivät pystyneet palauttamaan vaihtorahaa keskeytyneille setelitankkauksille. Vaihtorahan palauttaminen manuaalisesti kulutti asiakkaan asiakaspalvelu- ja kirjanpitoresursseja, joten asiakas halusi automatisoida vaihtorahan palauttamisprosessin säästääkseen resursseja. Tämä työ keskittyy uuden automatisoidun ominaisuuden suunnitteluun ja toteuttamiseen, ja pyrkii vastaamaan seuraaviin kysymyksiin: ”Kuinka paljon resursseja palautusprosessin automatisoinnilla voidaan säästää?” ja ”Mitkä ovat optimaaliset teknologiat uusien ominaisuuksien toteuttamiseen vanhan automaattitoteutuksen rajoitukset huomioiden?”

Työskentely alkoi tutkimalla muita ratkaisuja ja toteutuksia samankaltaisiin ongelmiin, jotta ongelmaan saataisiin laajempaa näkökulmaa ja ymmärrystä. Tutkimus johti alustavasti kolmeen lähestymistapaan ongelman ratkaisemiseksi. Näistä kolmesta lähestymistavasta valittiin käteispalautusten korvaaminen kuponkikoodien luomiseen ja lunastamiseen perustuvalla järjestelmällä. Valinnan jälkeen tarkempia suunnittelukysymyksiä pohdittiin asiakkaan kanssa, mikä johti järjestelmän käyttötapausten ja asiakasvaatimusten määrittymiseen.

Järjestelmä suunniteltiin kirjallisuutta ja online-lähteitä hyväksi käyttäen. Lopulliseksi suunnitelmaksi muodostui järjestelmä, joka tulostaisi muodostetun kuponkikoodin viivakoodina keskeytyneen setelitankkaustapahtuman kuitille. Tulostetun viivakoodin voisi lukea seuraavan tankkaustapahtuman aluksi, josta se välitettäisiin REST API:na toteutetulle Voucher-applicationille varmennettavaksi. Voucher-application varmentaisi kuponkikoodin tietokantaa vasten, ja varmennettu summa lisättäisiin käyttäjän aloittamaan tankkaustapahtumaan.

Optimaaliset toteutusteknologiat valittiin tehtyyn tutkimukseen pohjautuen. Kuponkikoodin sisällöksi valittiin GUID ja viivakoodin esitysmuodoksi QR-koodi. Voucher-application toteutettiin ASP.NET Core Web API:na, ja sitä suunniteltiin ajettavaksi asiakkaan automaattiverkostossa sijaitsevalla palvelimella. Ratkaisu toteutettiin.

Kirjoitusajankohtana uudet ominaisuudet oli otettu käyttöön pilottiversiona vain kahdella asemalla, jotka eivät olleet tuottaneet tarpeeksi tilastotietoja tarkkojen johtopäätösten muodostamiseen uusien ominaisuuksien toiminnasta. Tiedoista pystyi kuitenkin tulkitsemaan, että järjestelmä toimi halutulla tavalla, ja valitut teknologiat olivat tarkoitukseen sopivia. Pilottijakson aikana pilottiasemat olivat automatisoineet noin 4,75 tuntia rahanpalautustyötä. Tämä viittaisi siihen, että laajalle levitettynä toteutetulla järjestelmällä voitaisiin säästää huomattavasti asiakkaan resursseja.

Avainsanat: maksuautomaatti, vaihtoraha, viivakoodi, huoltoasema, .NET, REST

PREFACE

I would like to thank everyone involved in the Voucher-application project, especially my project manager Marko Felin, who helped me figure out the topic of this thesis and provided me with constructive and continuous feedback during the whole writing process.

I would also like to thank my examiners Professor Matti Vilkko and Project Manager Jari Seppälä for providing me with valuable guidance that helped me complete this thesis in a rightful manner.

Finally, I would like to thank my family and friends for the important support and feedback I received from them.

In Tampere, 26th October 2020

Ville Lepistö

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

API	Application Programming Interface
ATM	automated teller machine
BNA	banknote acceptor
CA	Certification Authority
CLI	Common Language Infrastructure
CLR	Common Language Runtime
CTS	The Common Type System
EF	Entity Framework
EPS	Electronic Payment System
GUID	Globally Unique Identifier
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure Sockets
IIS	Internet Information Services
IL	Intermediate Language
JSON	JavaScript Object Notation
OPT	outdoor payment terminal
PKI	Public Key Infrastructure
QA	Quality Assurance
REST	Representational state transfer
SCM	Service Control Manager
SDK	Software Development Kit
SSL	Secure Sockets Layer
TLS	Transport Layer Security
UI	user interface
UPC	Universal Product Code
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
WSDL	Web Services Description Language
XML	Extensible Markup Language

1 INTRODUCTION

The popularity of independent outdoor payment terminals (OPTs) has increased during the recent years. In 2018, the global market of OPTs was valued at 651.08 million USD in 2017, and by the end of 2023, it was expected to reach 1005.85 million USD [1]. As the popularity of all outdoor payment terminals has increased, independent outdoor refuelling terminals and unmanned gas stations have consequently also increased in popularity compared to the more traditional refuelling methods, such as refuelling the car up front and then afterwards paying to the cashier.

Due to the increased interest in using OPTs for refuelling, automotive fuel retailers and terminal manufacturers want to provide the best customer experience for the end user. It can be noted from the continuous implementation of new payment methods for OPTs, such as mobile applications and contactless payment cards. OPTs mostly work with advance payment, where the end user selects the initial maximum payment for the transaction which, for example with credit cards, will first be reserved as a cover reservation. After the refuelling process has been completed, the actual used amount will be charged from the end user. This process works well for all digital payment methods, but presents a challenge for cash payments, as OPTs are usually unable to return any unused cash in case of incomplete fillings.

This was a problem that a customer, a fuel retailer and an oil company, wanted to solve. Their current OPT system was unable to automatically return any unused cash from incomplete banknote transactions. The customer's current solution was to issue refunds manually after the end user has contacted their customer service. This approach presented issues for the customer. One issue was the amount of resources required from their customer service and accounting teams to manually return the unused money to the end users. Another issue was the poor user experience for the end user, who would have to manually contact the customer service in order to receive their change. Therefore, the customer required a new solution where returning the unused cash would be automated. This thesis mainly focuses on designing and implementing the required changes and software components within the constraints of the already existing system. The old system is also described to the level necessary to provide to a complete understanding of the implemented changes.

Various solutions for the cash returning problem had been presented before. Most solutions found were based on additional hardware capable of returning change. In a 1998 patent, an unattended system for selling and dispending fluids was described by Ram-

sey and Williams [2]. Compared to the customer's terminals, the patented system was also able to process coins. It was equipped with a separate change dispenser, which could return change accurately as both banknotes and coins. A similar solution was also presented by Gong [3]. However, this solution was capable of only processing banknotes.

Another solution for the problem had been presented by Miller, Jendges and Crynen [4]. The solution was intended for vending-type machines and was based on voucher codes. In case of an interrupted transaction, the machine would print out a voucher code to the user. The user could then receive a refund by presenting the voucher code to an automated money return machine in a central office [4].

Solutions combining hardware-based change dispensers and voucher codes were also found. A terminal system presented by Kurowski, Bruskotter and Swapp would issue a voucher code to the transaction's receipt in case it was interrupted before all the cash was used [5]. The user could then use this code at the terminal during a subsequent visit to add credit to their next transaction or input the code at a separate change dispenser, providing a method to return the remaining cash instantly.

In 2000, Wilson presented a solution to enhance cash transactions by using remote transponders [6]. In this solution, the terminals were equipped with a receiver, which could communicate with remote transponders. The transponders would be carried by the terminal users or be mounted to their vehicles, and they could be used to identify the user during a cash transaction. This provided a method to link the unused cash as credit to the remote transponders. The provided credit could then be redeemed at the user's next transaction [6].

1.1 Research questions and methodology

The main research questions this thesis seeks to answer are as follows: "What is the most suitable solution for the problem within the constraints of the old system?", "How much resources can be saved by automating the process of returning unused banknote payments?" and "What is the optimal voucher code format for the selected implementation?"

The general approach to this thesis was to first take a closer look on the existing research and solutions for similar problems to gain a better understanding of the problem and the possible solutions. This was done by researching existing patents and online sources. From this research, multiple methods of approach for a new solution were designed, and the most appropriate one was selected by its suitability to the customer's existing architecture. This general functionality of the solution was then designed with the customer based on their requirements and intentions for the system. This was done with the help of literature and online sources.

The literature used provided insight on the benefits and drawbacks of new technologies

and provided understanding of the technologies themselves. Technologies were investigated from books by Adams and Lloyd [7] and Vacca [8] for Public-Key Infrastructure, and a PhD dissertation by Fielding [9] for REST, among others. Literature was also used to gain insight on the processes of designing and implementing entire software components in an orthodox manner. Books by Reynders [10] and Massé [11], among others, were studied for this. The online sources were mainly used to provide the latest insight on technologies straight from their developers, and they provided answers on e.g. which frameworks to select for the implementations.

The technologies for the designed solution were then selected and the solution was implemented in practice. The applied solution was then compared to the old system based on the available data in order to see what benefits and possible drawbacks were achieved. The goal was to provide an accurate description of the entire development process and to help readers better understand what should be taken into consideration when designing and implementing a system with similar functionalities.

1.2 Structure

The thesis is structured in five main chapters. Chapter 2 covers the theoretical background for the technologies used in the solution and for those relevant in understanding this thesis. In chapter 3, the existing system components and functionality are described. The chapter also contains further details on the current issue, and customer's motivations and requirements for solving it. Chapter 3 then proceeds to conduct research on existing solutions to better understand the issue. Finally, the chapter focuses on selecting and designing the solution to solve the issue.

Chapter 4 describes the implementation process, chosen techniques and the challenges risen during the implementation process. It aims to reflect on how well the challenges and problems were accounted for in the design process. It also provides the final depiction of the implemented system. Finally, chapter 5 evaluates the results of the implementation. This final main chapter also discusses how well the project succeeded, and what future improvements and challenges lie ahead for the project. This chapter is followed by chapter 6, which concludes the thesis.

2 TECHNOLOGIES

This chapter contains descriptions of the technologies used and discussed in the scope of this thesis. The aim of this chapter is to provide the reader with the necessary understanding of the technologies to better understand the following design and implementation chapters.

2.1 .NET

.NET is an open-source developer platform developed by Microsoft, and it is intended to allow building different types of applications with different programming languages and cross-platform support [12]. Architecturally, it is composed by the following components: .NET Standard, .NET Implementations, .NET Runtimes, and .NET tooling and common infrastructure [13]. .NET Tooling and common infrastructure is the set of tools provided to the developer, including all the .NET programming languages (C#, F# and VisualBasic), their project systems and compilers. The .NET Standard provides the formal specification of the .NET Application Programming Interfaces (APIs), which each .NET implementation aimed for the specific version of the standard must implement. Microsoft's motivation for the .NET Standard is to establish greater uniformity in the .NET ecosystem, enabling developers to produce portable libraries usable across multiple .NET implementations, using the same set of APIs [14]. The architectural components are demonstrated in figure 2.1.

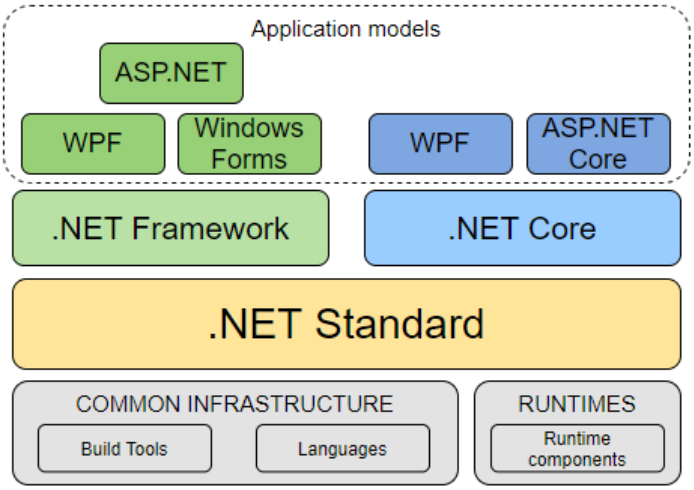

Figure 2.1. *Demonstration of .NET architectural components*

Microsoft actively develops and maintains four primary .NET implementations: .NET Core, .NET Framework, Mono and Universal Windows Platform, and each implementation consists of the base class library implementing the .NET Standard and at least one runtime component [13]. A .NET runtime is the component managing the execution of the program code, and acts as the execution environment for a .NET program [13].

2.1.1 Managed and unmanaged code

Microsoft divides code into two categories, called managed code and unmanaged code. Unmanaged code is essentially another term for compiled programming languages, such as traditional C or C++, where the code is compiled directly into machine code, leaving the programmer in charge of all memory management and security considerations [15]. Managed code is determined as code whose execution is managed by a runtime, which will automatically take care of compiling and executing the code, memory management, security boundaries and type safeties. Managed code as a term is fairly similar to interpreted programming languages, with the exception that it only covers programming languages that can be run on top of .NET [15]. Regardless of the programming language they were written in, managed code programs are compiled into Intermediate Language (IL) binary files, which the runtime is then capable of executing. The only exception to this rule is C++, which can be compiled directly into unmanaged machine code if the programmer so chooses [15].

Intermediate Language, also known as Common Intermediate Language [15], is a part of the specification for Common Language Infrastructure (CLI) standardized by ECMA International [16]. The standard specifies CLI as "a specification for the format of the executable code, and the run-time environment that can execute that code."[16]. The CLI provides a rich type system, The Common Type System (CTS), which is intended to support a wide range of different programming languages [16].

2.1.2 Common Language Runtime

Common Language Runtime (CLR) is one of the main runtimes in .NET and it, like the other runtimes, oversees running the IL binary files. It does this by reading and executing the IL from the binary files compiled from managed code and starts compiling it to machine code as the execution of the program continues. This process is called Just-In-Time compiling and it allows CLR to know exactly what the code is doing and how to best manage it. [15]. As managed code can be written in many programming languages that abide to CLI specifications, and then executed by the runtime, CLR provides multiple helpful features, such as cross-language integration and cross-language exception handling [17]. This makes designing cross-language components and applications easy, as objects and components written in different languages can communicate with each other

sharing the same runtime, allowing the programmer to choose the right tools for the right problems. CLI and CLR are demonstrated in figure 2.2

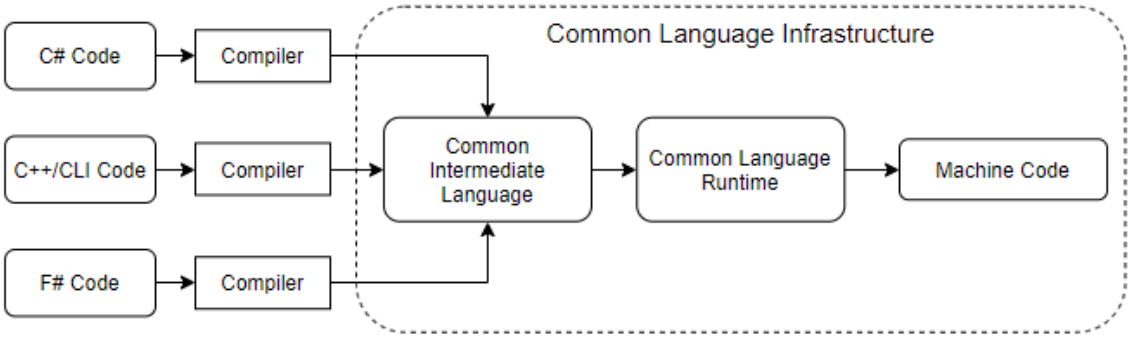

Figure 2.2. *Functionality of the Common Language Runtime*

CLR also supports interoperability, which means that the execution can pass from managed code directly into unmanaged code and back. This, for example, allows the programmer to wrap an unmanaged library and calling to it from the managed code [15].

2.1.3 C++/CLI

C++/CLI (C++ modified for Common Language Infrastructure) is a standardized superset of the standard C++ [18]. It adds CLI support to the language, allowing programmers to write managed C++ code which can be compiled to IL. This adds additional support for interoperability features that are not supported by other .NET languages [19], including the support for calling native functions from statically linked C++ libraries.

As native C++ libraries are unmanaged code, the data types returned from or passed to these function calls are not always directly compatible to CLI standard and must be converted accordingly to either match the CTS or to be supported by the native library. This data type conversion process is called marshaling, and C++/CLI has a built-in library to support these conversions [20].

2.1.4 .NET Framework

.NET Framework is Microsoft's first .NET implementation and has existed since 2002 [13]. Initially, .NET Framework was a stand-alone development framework and did not support the .NET architecture described in section 2.1, but after it had been defined, support for the initial .NET Standard was implemented in version 4.5 of .NET Framework [14]. At the time of writing, the newest released version is 4.8 [21], implementing version 2.0 of the .NET Standard [14].

.NET Framework consists of the Common Language Runtime and the .NET Framework

class library [22]. The class library is an object-oriented collection of types that implement the targeted .NET Standard specification and many additional functionalities. Programmers can use these provided types to derive the functionality for common programming tasks, like string management, file access and database connectivity, to their own managed code solutions [22]. .NET Framework also provides types to provide support for developing different types of apps and services, including, but not limited to: Console applications, applications with graphical user interfaces (Windows Forms, Windows Presentation Foundation), Web applications (ASP.NET) and Windows services [22].

Even though .NET Framework provides a vast class library, it is not always the optimal .NET implementation to choose, as it does not provide cross-platform support and can only be run on Windows systems [23]. However, the large class library and its native inclusion in most Windows operating systems [21] makes it a great solution for developing Windows-based applications.

2.1.5 .NET Core

.NET Core is a cross-platform implementation of .NET, capable of running on Windows, macOS and Linux [13]. It has support for many kinds of apps, including console apps, serverless functions in the cloud and web APIs [24]. It was designed to handle server and cloud workloads at scale [13]. While having cross-platform support, .NET Core allows the usage of platform-specific capabilities, such as operating system APIs [24].

The newest evolution of .NET Core is the .NET 5, which will be the main .NET implementation in the future [25]. It implements the newest version of the .NET Standard, .NET Standard 2.1 [14].

2.2 Windows Service Application

Microsoft Windows Service is a type of program supported by the Windows operating systems. Services operate in the background in their own Windows sessions and do not show any user interfaces. This makes them ideal for server usage and for long-running functionality that should not interfere with other users working on the same computer [26]. Services can be configured to start automatically along with Windows without requiring any user logins. Services also run in their own security context of a specific user account, which can be determined in the service configurations [26].

Before services can be used, they must be installed and loaded into the Service Control Manager (SCM). The SCM, among other things, maintains a database of installed services, maintains the status information of running services, and starts services either upon demand or system start-up [27]. The SCM also transmits control requests to running services. Once a service application has been installed and loaded into SCM, it

can exist in three basic states: Running, Paused or Stopped [26]. The first state indicates that the process is running normally, while the second indicates that the process is still running, but the functionality has been suspended. The actual functionality for the Paused-state is defined in the service application itself, and it can vary. The third state indicates that the process is not running at all. In addition to the three basic states, the service process can exist in transitional states between the basic states [26].

In addition to controlling services from SCM Management Console, Windows Services can be controlled programmatically with more precise commands via the ServiceController-class, available natively in most versions of the .NET Framework [28].

2.3 Barcode

A barcode is a pattern that is encoded in defined dimensions using bars and spaces [29], and the basic concept of a barcode has existed for a long time. In 1949, Woodland and Silver filed a patent for a "Classifying apparatus and method" [30]. The patent described patterns that are automatically classifiable by photo-sensitive apparatus. The patterns used as an example were white lines on a dark background, mostly containing four lines. The first line from the left is the datum line, and the position of all the other lines is fixed with respect to the datum line. All the other line positions were assigned a corresponding number based on the binary number system, the last one representing 2^0. In an example four-line pattern, the first line would be the datum line, while the second, the third and the fourth line locations would represent 2^2, 2^1 and 2^0, respectively. If the line location would have a white line, the represented number is added to the final value of the pattern. If the location would not contain a line, the represented number is not added to the final value. A four-line pattern is demonstrated in figure 2.3.

Figure 2.3. *Examples of Woodland and Silver's four-line barcode patterns representing the decimal values of 7 (left) and 5 (right) [30].*

The help understand this, the patterns in figure 2.3 can also be thought as binary representations of the numbers, where the first line is the datum line and can be ignored while forming the number. Analysing the rightward pattern, there is a white line representing the number 1 in the second and the fourth line locations, while the third location does not have a line, representing a 0. This forms the binary number 101, which can be converted

to a decimal value as follows:

$$101_2 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 5_{10}$$

Similar process can be performed to the leftward pattern in the figure, resulting in the decimal number 7.

The Woodland and Silver patent also described an example of circular pattern format containing ten line locations, to provide improved readability and representation of larger values [30].

After the initial barcode invention, various methods of encoding information in barcodes were developed for various purposes. These methods are known as symbologies [29]. The first generation of barcodes consisted of linear barcodes, utilizing lines and spaces that follow patterns, similarly to the Woodland and Silver design. These included symbologies such as Codabar, ITF-14, Code 39, Code 128, and most notably, the Universal Product Code (UPC), which would end up revolutionizing world-wide commerce with its adaptation in supermarkets [31]. However, on the linear symbologies, this thesis will focus on Code 128 due to its relevance in section 3.4.2.

2.3.1 Code 128

Code 128 is a barcode symbology standardized as ISO/IEC 15417 [29]. It is a continuous barcode with no maximum length and with a high-density of 11 modules per data symbol character [29]. A module is the smallest width of a bar or a space in the barcode format. The encodable character set of the symbology includes all 128 ISO/IEC 646 characters, 4 non-data function characters, 4 code set selection characters, 3 different start characters and a single stop character [29]. The barcode pattern itself consists of six sections: a leading quiet zone, a start character, one or more characters representing the actual data and special characters, a symbol check character, a stop character and a trailing quiet zone [29]. Each individual symbol character in the barcode consists of 3 bars and 3 spaces, apart from the stop character, which consists of 4 bars and 3 spaces [29]. This structure is demonstrated in the figure 2.4.

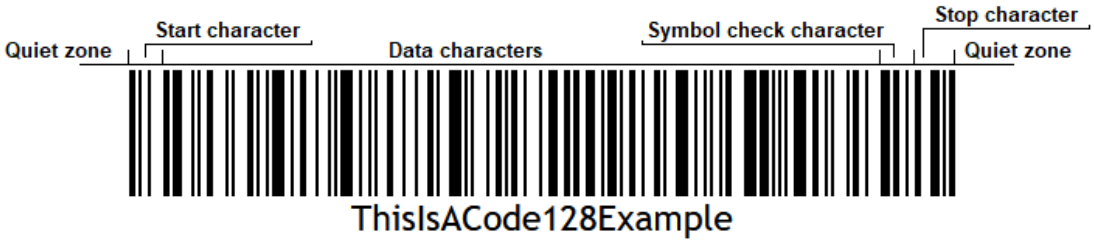

Figure 2.4. *An example of a Code 128 barcode*

Due to the specification on the symbol structure, Code 128 can only represent 103 data

characters, 3 different start characters and the stop character. To allow the encoding of all 128 ICO/IEC 646 characters, Code 128 contains three different code sets of 103 data characters, where the same symbols reflect different characters: 128A, 128B and 128C. The code set initially used depends on the selected start character, and it can later be switched in the data character section with the use of a code set selection character specified in the code set. The code sets include the following characters: The code set 128A contains ISO/IEC 646 characters 00-95 (control characters, special characters, 0-9, A-Z), 4 non-data function characters and 4 code set selection characters. In turn, the code set 128B contains ISO/IEC 646 characters 32-127 (special characters, 0-9, A-Z, a-z), 4 non-data function characters and 4 code set selection characters. 128C was intended for a compact representation of numbers, and only contains numbers from 0-99, 2 code set selection characters and a single non-data function character [29]. The compact number representation is achieved by encoding double digit numbers in a single symbol character. The example in the figure 2.4 only uses the code set 128B, as it contains both upper- and lowercase letters and does not contain a large amount of consecutive numbers or any control characters. The encoding abilities of the character sets are represented in table 2.1.

Table 2.1. *Character encoding ability of the different Code 128 character sets*

	128A	**128B**	**128C**
ISO/IEC 646 Control characters (e.g. NUL, LF, CR)	x		
ISO/IEC 646 Special characters (e.g. !, ", #, $, ?, *)	x	x	
0-9	x	x	x
A-Z	x	x	
a-z		x	

In addition to linear barcodes, another type of barcode symbologies, known as matrix barcodes, has been developed. Matrix barcodes are also known as two-dimensional barcodes, as they represent data both vertically and horizontally. Matrix symbologies, especially the QR code, have recently started to grow more popular [32].

2.3.2 QR Code

QR Code, or Quick Response code, is a matrix barcode symbology, which consists of square modules arranged in a square pattern. QR Code also includes unique finder patterns at three corners of the pattern, intended to assist in easily locating its position, size and inclination [33]. QR code symbol size and contents are represented in versions, and the minimum size for a QR Code symbol is version 1, which consists of 21x21 modules. versions go up in increments of 4 modules, and the maximum size for a QR Code symbol is 177x177 modules, named version 40 [33]. An example QR Code symbol is presented

in the figure 2.5.

Figure 2.5. *An example of a QR Code symbol*

QR Codes can encode purely numeric data, alphanumeric data, 8-bit byte data using the ISO/IEC 8859-1 character set, or Kanji characters [33]. Each type of data has its own density, and the same version of a QR code can encode different amounts of data in different characters. The maximum numeric data for a version 40 symbol is 7089 characters, whereas for kanji it is only 1817 characters [33]. However, symbols can also be encoded in mixed mode, which in some cases allows for a larger character amount when using different character types [34].

QR Code symbols also have four selectable Reed-Solomon error correction levels: L, M, Q and H [33]. These levels allow for the recovery of approximately 7%, 15%, 25% and 30% of the symbol codewords, respectively. However, they also take up character space from the encodable data. A version 40 QR Code with error correction level L can encode up to 7089 numeric characters, whereas the same Version 40 code with error correction level H can only encode 3057 numeric characters [34]. Error correction level is often indicated along with the version information, e.g. version 40-L.

Like a Code 128 barcode, a QR Code symbol must be surrounded by a quiet zone border. The quiet zone is one of the function patterns of the symbol. Other function patterns include three finder patterns, timing patterns and the alignment patterns in case of symbols larger than version 2 [33]. Function patterns encode no actual data, and they are required for the identification of the symbol's characteristics to assist in decoding [33]. The data is instead stored in the encoding region of the symbol. The encoding region is masked with a pattern to provide a well-balanced layout of black and white modules to improve reliability[33]. It contains the format information, the actual data and error correction codewords, and version information for QR Code symbols of version 7 or larger [33]. These sections of the earlier example QR Code are demonstrated in the figure 2.6.

Figure 2.6. *QR Code structure*

As can be seen from the picture, the example QR Code has the required function patterns. The three finder patterns indicate the direction in which the QR Code is printed, and they must always be surrounded by a one-module wide separator border [33]. The two timing patterns of alternating black and white modules enable the determination of the symbol density and version, and provide datum lines for coordinates in the symbol [33]. As the example is version 5-H, one alignment pattern is present to help with the orientation. The symbol is also surrounded by the quiet zone.

The only highlighted encoding region section in the example is the format information. The format information contains the mask pattern of the symbol, along with information about the error correction level and locations [33]. All the non-highlighted modules of the example hold the actual data and error correction codewords of the symbol. The version information section is not present in this example, as it is smaller than version 7. However, when present, the version information would contain the version of the QR Code symbol and more information about the error correction [33].

2.4 Representational state transfer

Representational state transfer (REST) is a software architectural style introduced by Fielding in 2000 [9]. It describes a set of software engineering principles and constraints used for designing distributed hypermedia systems. REST-compliant systems are called RESTful and must fulfil six constraints.

2.4.1 Constraints

The six constraints a RESTful system must abide by are the following: Client-Server, Stateless, Cache, Uniform Interface, Layered System, Uniform Interface and finally, as an optional constraint, Code-On-Demand [9].

Client-Server

The first constraint in REST is the client-server architectural model. This model aims to separate the design and development concerns for data storage and user interfaces by dividing the systems into components. Separating the client-side from the server-side improves scalability and portability for multiple platforms [9]. The components communicate via an API, and they do not need to know about each other [35]. This model allows both sides to be developed more independently and greatly simplifies the server-side, making it more scalable for bigger solutions [9].

Stateless

The second constraint requires the client-server interaction to be stateless. This means that each request from a client to the server must contain all the information necessary for the server to understand the request. If a session is required for the system, its state is stored entirely on the client-side [9]. Stateless-ness incudes the properties of reliability and scalability [9]. Scalability is increased as the server does not have to store the states of any client between the requests, allowing it to release resources quickly and simplifying the implementation process. Reliability is improved, as it is easier to recover from occasionally failures because only one component knows the state of the system, preventing state-mismatches.

This constraint also presents a trade-off. As data cannot be left on the server, it must be sent every request when it is needed. This might negatively affect network performance due to repetitive data being sent [9].

Cache

To improve the network efficiency, the cache-constraint was added [9]. The data in the server responses must be defined as cache-able or not. The client can then save this data to memory and use it later instead of sending equivalent requests, removing the requirement for repetitive data requests. However, this might also decrease reliability, in case stale data would differ from the data that would be directly obtained from the server [9].

Uniform Interface

The fourth constraint of REST is the uniform interface. The uniform interface means generalizing the component interface, separating the interface for the provided services from the implementations of clients and servers. This comes at the cost of efficiency, as the standardized form of uniform information must be converted to suit the needs of the specific applications [9]. However, the uniform interface simplifies overall system architecture, improves the visibility of interactions and encourages the independent evolvability of the services [9].

Layered System

A RESTful system can have multiple layers. The layered system constraint dictates the component behaviour by allowing each component to only have knowledge of the immediate layer it is interacting with. This allows the system architecture to consist of hierarchical layers. As a client is not able to identify which layer the information in the response originated from, this enables transparent deployment of intermediaries, e.g. load balancers, between a client and server using the uniform interface [11].

The layered system constraint places a bound on the overall system complexity, allows the simplifying and optimizing of the layered components, while also improving system scalability [9]. Compared to a non-layered system, the main disadvantage of the layered system is the additional overhead and added latency to the processing of data due to it passing through multiple components.

Code-On-Demand

The final constraint of REST is the code-on-demand style, which allows the server to temporarily transfer executable code, e.g. JavaScript, to the clients. This can simplify the client requirements, reducing the number of features to be pre-implemented by allowing server to extend the client functionality. However, code-on-demand is only considered optional because this creates a technology coupling between the server and clients, as the clients must be able to understand and execute to code being transferred [11].

2.4.2 RESTful web services

A web service is a web server purposefully built for supporting the needs of a web site or some other application [11]. Web services provide APIs enabling clients to communicate with them. The Web APIs act as the interface between the server's functionality and the clients' requests. Web services implemented using the REST architectural style are

called RESTful web services. This thesis only focuses on RESTful web services based on the common Hypertext Transfer Protocol (HTTP).

A HTTP-based RESTful web service implements a REST API acting as the interface between the client and the server. A REST API uses a Uniform Resource Identifier (URI) to address its resources, in the form of a URL. The URI indicates which resource is being accessed by the client. To perform any operations on this resource, a HTTP method is also required in the request. The methods commonly used are GET, POST, PATCH, PUT and DELETE [35], which dictate the behaviour of the request. According to Ahmad, GET should be used for retrieving resources without altering them [35]. POST should be used for creating new resources to the server, while DELETE should be used for removing them. PATCH and PUT should be used for updating existing resources. To be more specific, PATCH is used for partial updates and PUT for updating the entire resource. The server will respond to these requests with accordant HTTP response codes and the requested content, which is typically formatted in JavaScript Object Notation (JSON) or Extensible Markup Language (XML) data-interchange formats [35]. RESTful Web APIs have many more implementation conventions and rules [11], but they were left out of the scope of this thesis. A HTTP request-response pair to a RESTful Web API is demonstrated in figures 2.7 and 2.8.

Request

```
GET /api/users HTTP/1.1
HOST: testServer:8000
Accept: application/json
```

Figure 2.7. *HTTP GET to /api/users*

Response

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Server: Microsoft-IIS/10.0

[{"name":"John","id":1},{"name":"Matt","id":2}]
```

Figure 2.8. *The corresponding HTTP response containing a JSON-formatted list of users*

The RESTful Web API implementation style has many benefits. As REST emphasizes on fixed conventions of implementation, users have an understanding of how each REST API is likely constructed [35]. This is different compared to some other web service alternatives, where no conventions are present, and the services provided by the API are described with the use of Web Services Description Language (WSDL). Other advantages of the REST APIs include the previously mentioned separation of development concerns of client and server, simplicity and reliability of the stateless-ness, and the scalability of the systems due to the uniform interface and layered system architecture.

2.5 Public-Key Cryptography

Public-Key Cryptography refers to a form of cryptography where two different keys are used to encrypt and decrypt content. These two keys are called a key pair, and the keys in it are called the public key and the private key [36]. The public key is made publicly available for anyone wanting to securely exchange information with a certain entity. The private key is kept a secret known only by the entity in question. Each side then uses their available key to either encrypt or decrypt the information exchanged. The keys in the key pair are mathematically related to each other, as one key must be able to decrypt the data encrypted by the other [7]. However, as one of the keys is publicly available, the keys must also be adequately different to the point where knowing one key does not allow the computation or derivation of the other in any practical time frame, even with a lot of computational power [7]. An encrypted message sent using public-key cryptography is demonstrated in figure 2.9.

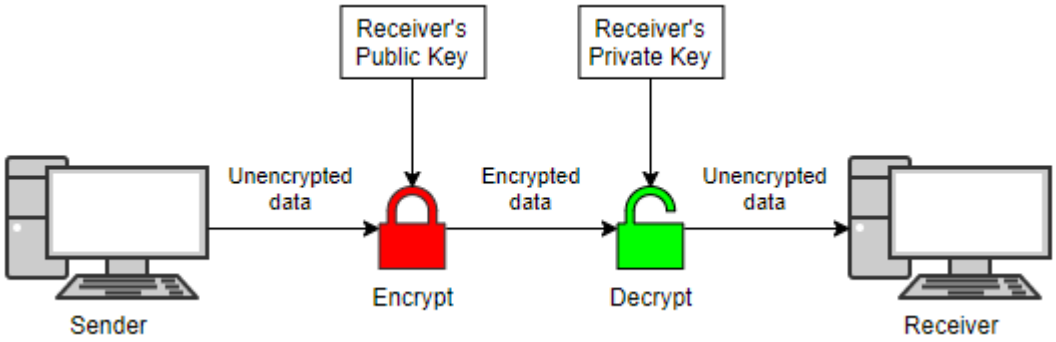

Figure 2.9. *Illustration of Public-Key cryptography*

One of the main benefits of public-key cryptography is the enabling of secure communications between strangers in a large scale [7]. However, in order to achieve this, the public-key technology must be made available in a uniform manner for multiple applications and environments [7]. A security risk is also involved, as the public keys used can be substituted by malicious individuals. Therefore, the entity using the public key to encrypt information cannot be certain that the used key belongs to the intended recipient. These, among other issues, led to the concept of a Public Key Infrastructure (PKI).

2.5.1 Public Key Infrastructure (PKI)

A PKI is a framework employing public-key cryptography and digital certificates for secure communications [37]. It is generally used for encrypting data, digital signatures and validating data integrity [7]. It is the technology providing security for Secure Sockets Layer (SSL) and Hypertext Transfer Protocol Secure Sockets (HTTPS) protocols, used to conduct secure E-commerce over the Internet [8]. SSL was later replaced by Transport

Layer Security (TLS) and is considered entirely deprecated as of 2015 [38].

A PKI consists of different programs, procedures and security policies. It is an infrastructure used for identifying users, distributing and maintaining encryption keys and enabling technologies to communicate via encrypted communications [37]. It is also used for creating, distributing, maintaining and revoking certificates [37]. A PKI relies on a trusted third party to validate the integrity and ownership of public key received from any entity [36]. This trusted party is called a Certification Authority (CA). A CA typically issues encrypted digital certificates to confirm the identity of the subject and then binds the identity to the public key contained in the digital certificate [36]. The CA then uses their own private key to encrypt this certificate. The issued public key is then available to all interested entities in the form of a self-signed CA certificate [36]. These CA issued certificates can then be validated by using the trusted public key of the CA.

A PKI requires life-cycle management for the keys and certificates, which consists of three phases: Initialization, Issued and Cancellation [7]. Before a PKI can be used, entities must first be initialized into it. This phase includes the registration of the entities, generating key pairs, creating and distributing the keys to the registered entity, and disseminating the generated certificate to other entities in the PKI [7]. Initialization phase might also include a backup of the key and certificate by a trusted third party [7].

Once the initialization is completed and the certificate is issued, further life-cycle management is required. This includes certificate retrieval and validation, and key recovery and updating [7]. Certificates must be readily available to encrypt data or verify signatures of other entities. Key recovery is required in case the private keys are lost, in order to prevent permanent loss of possibly critical information. As certificates are issued with a fixed lifetime, a new public/private key and certificate must be issued every time the certificate nears its expiration [7].

The last phase of the life-cycle management is the cancellation phase. It consists of certificate expiration, certificate revocation, maintaining a history of the issued keys [7]. When a certificate expires, it must be either renewed with the same keys or updated with a new key pair. If the entity is no longer in the PKI, no action will be required. Certificates might also have to be revoked for various possible reasons, like a suspected private-key compromise. A key history is required in order to provide decrypting of stored material encrypted with an older certificate [7].

2.5.2 Certificates

The key to PKI is the digital certificates. The public-key certificates are used to verify data integrity and guarantee that the public key belongs to the claimed entity [7]. They are used to bind entity's name and possible other attributes to the corresponding public key [7]. The current preferred standard for digital certificates is the X.509 standard [37].

The X.509-standard identifies multiple fields and values to be used in the certificates. It currently has three versions [39]. Version 1 contains the most important basic fields of the certificate. Microsoft lists the fields as follows [40]:

- Version, specifying the version number of the certificate.
- Serial number, unique value assigned by the CA.
- Signature algorithm, specifying the algorithm used by the CA to encrypt the certificate.
- Issuer, distinguished name of the CA.
- Validity, the time interval in which the certificate is considered valid.
- Subject, the distinguished name of the entity associated with the public key in the certificate.
- Public Key, the public key contained in the certificate.

These fields help guarantee the authenticity of the certificate to a reasonable degree. Following versions of the standard extend the available information by adding additional fields to the certificates to correct deficiencies associated with the older versions [7].

2.6 Internet Information Services

Internet Information Services (IIS) is a Web server software developed by Microsoft, designed to host multiple types of content on the Web. According to W3Techs, it is currently the fourth most popular web server in use [41]. Microsoft intends it to be flexible, secure and easily manageable with its scalable and open architecture [42]. IIS is based on a modular architecture, which means that all the Web server features are managed as standalone components which can be added, removed or replaced [43]. This has several advantages. Removing unused server features improves the server security by reducing the attack surface area, while also improving performance and reducing memory footprint [43].

IIS is also extendable, allowing developers to build new server components that extend or replace existing server features [43]. This includes custom authentication schemes, monitoring, load balancing and state management [43]. Extensions can be developed using native C++ or .NET using the ASP.NET web framework mentioned in section 2.1.4.

IIS supports multiple popular web applications frameworks, including ASP.NET, PHP, ASP.NET Core [44][45]. It also allows web sites to use more protocols than HTTP and HTTPS [46].

3 MOTIVATION AND DESIGN

The aim of this chapter is to first describe the functionality of the current payment terminal system, its most relevant subsystems, and to shed light on the problem the customer is having. This chapter then takes a look at some other existing solutions available, which were beneficial to solving the problem. After this, a general approach for the solution is chosen based on the customer's current system, requirements and the research on other payment terminal implementations. Lastly, this chapter further elaborates on the design process of the selected solution.

3.1 Current implementation

The basic concept of the OPT system is as follows: The end user drives their car to the terminal, parks and proceeds to use the refuelling terminal with some form of a user interface (UI). The UI can be another system external of the physical terminal, like a mobile application, or the UI of the physical terminal device. User then selects the desired payment method and the monetary amount they are willing to pay for the refuelling, after which the selected payment method is verified to be able to pay the selected amount. After the payment method has been authorized, the user can begin refuelling on a pump connected to the payment terminal by picking up a fuelling nozzle and starting the refuelling process, which continues until the user has refuelled for the total sum they paid for or places the nozzle back to its holder. The payment process is then finalized by charging the user for the actual sum they refuelled for. As a final step, the user can use the terminal to print a receipt for the transaction. This process is represented in the figure 3.1.

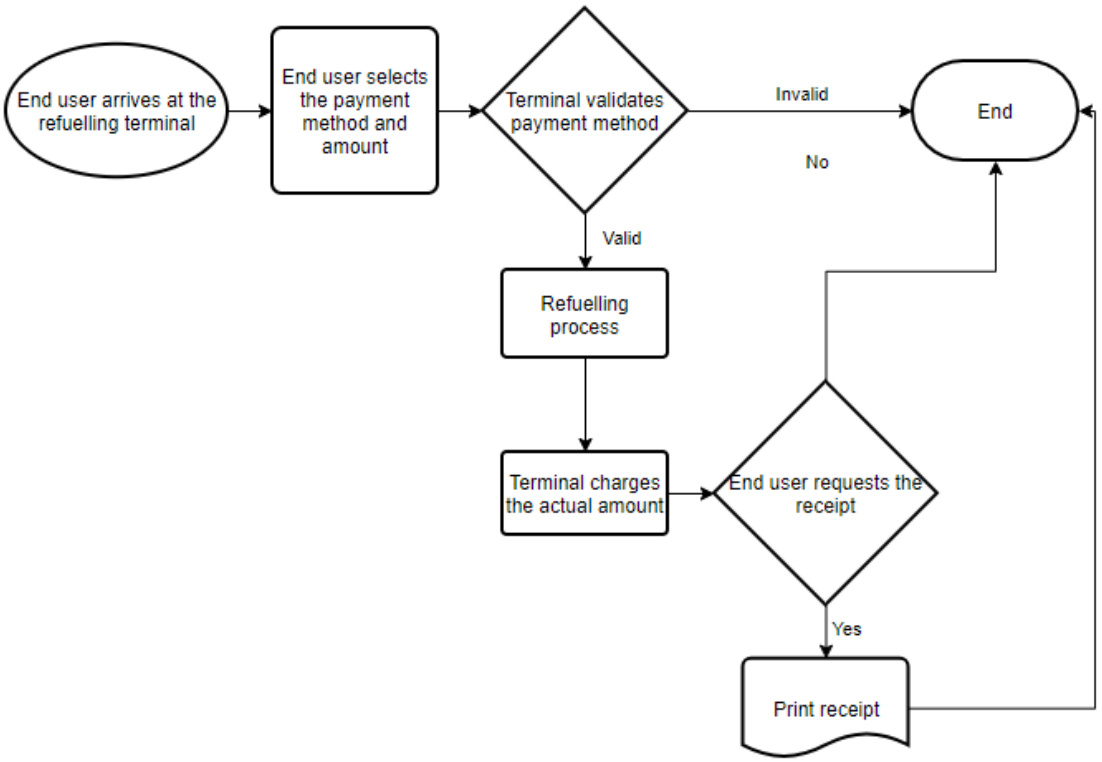

Figure 3.1. *A flowchart representing terminal functionality*

The current system consists of many components, but the scope of this thesis mainly covers only two of the software components, and the physical terminal device.

3.1.1 Payment Terminal

The payment terminal has support for several input peripherals, of which the following are relevant to this thesis: Card reader, banknote acceptor (BNA), receipt printer and barcode scanner. The card reader is capable of reading contactless, chip and magnetic stripe cards, and the BNA is configured to recognize all of the most common Euro banknotes. The receipt printer is a standard thermal printer, capable of printing the most common barcodes, while the barcode scanner is a standard two-dimensional scanner capable of reading most barcode formats. The payment terminal also has a display screen and several control buttons.

For controlling these peripherals, the payment terminal provides a Software Development Kit (SDK) in the form of statically linked native C++ libraries. This SDK is used by OPT-Sequencer, the main software component in the system.

3.1.2 OPT-Sequencer

OPT-Sequencer is the software component in charge of handling most of the payment terminal's functionalities. It is responsible for monitoring the states of all the input peripherals and other components in the payment terminal, and raising alarms to the monitoring systems in case of component failures. OPT-Sequencer also controls and monitors the software component responsible for controlling the physical pump with an SDK, like the one provided by the payment terminal.

OPT-Sequencer is implemented as a Windows Service Application and developed using .NET Framework. It is mainly written in C#, but it also contains C++/CLI code to add support for the required interoperability for calling methods from the payment terminal SDK.

As the main component in charge of other components, OPT-sequencer is also responsible for controlling the terminal's main user sequences. It detects user's payment inputs and starts the appropriate payment sequences based on the received inputs. Except for banknotes, which are physically detected and validated by the BNA, OPT-Sequencer verifies the payment inputs using external software components, known as Electronic Payment Systems (EPSs). The system has multiple EPS components, but the one relevant to this thesis is responsible for processing customer's fleet cards.

3.1.3 EPS-Component

Like OPT-Sequencer, EPS-component is also implemented as a Windows Service Application, using .NET Framework and written in C#. The EPS-component oversees validating the customer's custom payment methods, including their own credit cards, gift cards and other payment methods.

For initial card payment requests by the OPT-Sequencer, the EPS-component validates the payment method by verifying multiple different payment rules, such as card expiration or possible product restrictions. If all the requirements for card validity are met, the EPS-component then validates the payment method for the monetary amount selected by the customer via an external payment verification service. If the requested amount is approved, the EPS-component sends the result as a response to the OPT-Sequencer. EPS-Component also adds the possible payment method related product discounts to the response message. This process is called pre-authorization.

For finalizing the initial payment validation, the OPT-Sequencer sends a payment finalization request with the actual final details of the completed transaction. EPS-component then notifies the payment verification service of the actual payment amount of the transaction, finalizing the transaction with the correct sum. The whole refuelling sequence is illustrated in the figure 3.2.

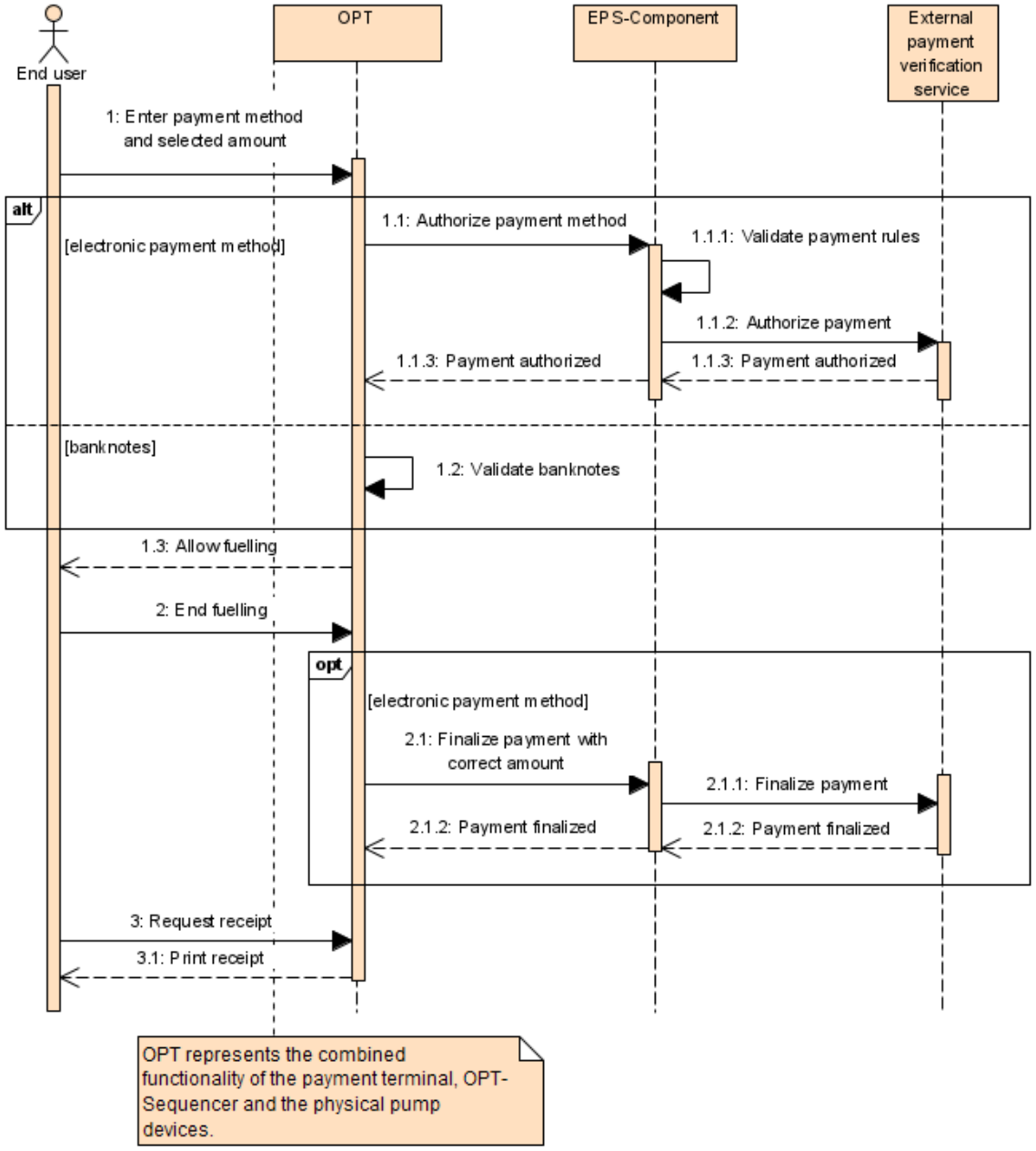

Figure 3.2. *A simplified sequence diagram representing a single successful refuelling*

The sequence diagram represents a successfully completed refuelling sequence. To simplify the diagram, the functionality of the payment terminal, OPT-Sequencer and the pump devices has been merged to a single entity.

3.1.4 The banknote issue

As the payments are processed in two parts, pre-authorization and finalization, handling an incomplete or erroneous refuelling is done automatically for electric payment methods during the finalization phase. However, as can be noted from the figure 3.2, incomplete refuellings cause a problem for cash payments. Banknotes are only processed by

the BNA in the beginning of the transaction. Because the payment terminal is unable to dispense any change, the actual payment amount cannot be corrected at the end of the transaction. This presents a problem where transactions aborted by large initial payments or possible pump errors cannot be refunded to the end user directly at the payment terminal.

The customer's current solution for refunding incomplete cash payments depended on the station in question. If the station is a manned station, the refund could be handed out by the personnel at the station. However, for unmanned stations, the end user would have to contact the customer service, who would validate the need for the refund. After validation, the refund would be handed out as a bank transfer. The customer observed these refunds for a one-month observation period. According to the data, there was a total of 716 banknote transactions where the entered monetary amount was greater than the final price of the transaction. During the observation period, customer's accounting department issued about 200 payments to the end users, each taking approximately 15 minutes to process. About 160 of these transactions had a difference of more than 5 euros. For the customer, the refunds caused additional labour costs and possible negative publicity resulting from bad end-user experience.

Because one refund payment takes around 15 minutes to process, customer's accounting department spent approximately 3000 minutes, or 50 hours, processing refund payments during the observation period. This results in approximately 6.7 man-days of additional labour required in one month. In addition, 716 incomplete banknote transactions during a one-month period signals that there are multiple end users with a negative experience, which could lead to negative public exposure. In order to solve these issues, the customer required an automated solution to make the cash refunding process more effective for labour and other costs, while also improving the end-user experience of the payment terminals.

3.2 Other existing implementations

To better understand the issue and different ways to resolve it, research on other existing solutions for similar problems was conducted. As unmanned OPTs have been commonplace for a long time, many solutions already have already been presented. The most patented solutions found for fuelling terminals were different kinds of hardware-based solutions. In a U.S. patent from 1998, Ramsey and Williams described an unattended automated system for selling and dispensing fluids [2]. The system was intended for the dispensing of motor fuels, and it could accept payments with both cash and cards. For cash payments, the patent described the system to have a cash acceptor and a change dispenser, which both were able to process both coins and banknotes. The aim of the system was to reduce labour costs by handling change dispensing automatically and to allow virtually unattended operation of a service station [2]. Another similar hardware-based solution was also described by Gong in another patent [3]. However, the system

described in this patent could not dispense change as coins, as the system only supported banknotes for cash payments.

Another type of solution was presented in the U.S. patent 5,055,657 by Miller, Jendges and Crynen [4]. The patent described a vending-type machine capable of printing a voucher code in case of an interruption during the purchase process. If a payment is interrupted, a voucher code would be printed to the user, as both plain-language and machine-readable. The same code would also be transferred to the memory of a central computer in a central office via a data communication line. In order to receive the refund, user could then present the voucher to an automated money return machine in the central office. The system would have a possible upper monetary limit for issuing vouchers in order to prevent the vouchers from becoming too valuable. As a clear benefit for this solution, the patent states that the components required could be installed without difficulty to already existing vending-type machines [4].

Another common approach were solutions combining the hardware-based change dispensers and voucher codes. These solutions functioned by printing voucher codes that could later be used on the terminal for a new transaction or changed to cash on a separate cash dispenser. In a 1997 patent filing, Kurowski, Bruskotter and Swapp describe an existing system capable of issuing credit voucher codes for the remaining balance [5]. The terminal user can then enter the code into a keypad on the pump system and be granted the previously unused balance on a subsequent visit to the terminal. Kurowski, Bruskotter and Swapp further expanded this system by incorporating a separate change dispenser to the system. The change dispenser could be located either at the pump or at a separate location. If the user has unused balance left at the end of the transaction, the voucher code would be printed on the receipt, while the code and transaction information would also be transmitted to the memory of the change dispenser. User is then able to make the choice between cashing out the voucher code instead or using it on a subsequent visit.

A similar solution was described by Enright in the U.S. patent 6,845,907 [47]. The patent described an apparatus providing cash or printed vouchers to users in self-service environments. The apparatus would be able to print out vouchers for predetermined amounts specified by card users, to allow withdrawal of cash from a service provider located nearby, instead of going to an automated teller machine (ATM). The patent also stated that in alternative embodiments of the invention, the apparatus could include a cash accepting device and a cash dispensing mechanism. This would then allow to user to pay with cash and receive the unused change as either cash from the dispensing mechanism or a printed voucher. The voucher could then be redeemed as cash or used for purchasing goods at the service provider. The main benefit listed for the system was to provide users with easier access to cash than going to an ATM, which could then encourage them to perform additional purchases at the service provider [47].

An entirely different type of approach for returning change at fuelling terminals was produced by Wilson in 2000 [6]. The invention described in the patent is an automated fuel

dispenser for enhancing cash transactions. The fuelling system would be equipped with a receiver to communicate with remote transponders, which would be carried by users or be mounted to their vehicles. The data received from the transponders could then be used to identify the user, which is otherwise difficult in a cash transaction. This identity information could then be used to award loyalty benefits or pay back unused change as credit to the user now identified by the transponder. As additional benefits for the system, the patent states enhanced marketing efficiencies and improved safety in the fuelling environment [6].

As voucher codes were observed to be a popular form of dispensing change, research was also conducted on other voucher code systems unrelated to fuelling terminals. In 1998, an automated voucher system for providing employee cash pay-outs was described by Gilbert, G. A. Welstad and T. Welstad [48]. The system was designed to help employment contractor employees, who work for short periods of time for other companies hiring the worker from the contractor. As they usually work for a short time, a direct deposit payment for the worker was rarely set up by the hiring company, and the employees were mostly paid by bank drafts. According to the patent, bank drafts were determined to be a difficult method of payment for some employees, and an automated cash-out voucher system was planned [48]. The system would work by having a service terminal and a cash-out machine. Employees would first use the service terminal to select between a cash voucher and a bank draft. If the employee would choose a cash voucher, the terminal would verify the request against possible limitations, issue the voucher and record the payment data in a database. The employee would then use the nearby cash-out terminal, which would verify the inputted voucher from the database, perform the cash disbursement and flag the voucher as cashed-out.

Another voucher-based system was disclosed in a 2003 patent by Luciano [49]. The system was intended to enable more use for non-cash-based inputs in gaming systems for games of chance. The patent describes a group of interconnected gaming and cash exchange terminals. In one embodiment of the system, it would work by the player first using a cash exchange terminal to insert money, which would then be validated by a BNA. After the terminal has validated the money, the transaction is then recorded to a central terminal and a barcode voucher identifying the transaction is printed. The player would then be able to scan this code at any gaming device connected to the system and use the paid transaction to play the game. When the player terminates the play, a new voucher code would be similarly generated in case any balance is left. The player would then be able to use the new voucher to continue playing on another game or use a cash exchange terminal to cash-out the voucher.

3.3 Selecting the solution

After researching other ideas and solutions for the problem, three different methods of approach were planned: A hardware-based solution, a voucher-based solution and a

loyalty card-based solution. The hardware-based solution would be similar to the one described by Ramsey and Williams [2]. It would include some mechanism for dispensing cash to the end user immediately after the transaction has completed in case a change pay-out is required. This would provide the end user with direct access to cash and reduce the need for further actions by the customer.

The voucher-based solution planned was a combination of the systems described by Miller, Jendges and Crynen [4] and by Kurowski, Bruskotter and Swapp [5]. In case of an incomplete banknote fuelling, the system would print out a voucher code on the receipt of the transaction. This code and other necessary transaction information would then also be transferred to a central database located somewhere in the customer's terminal network. For their next transaction at any of the customers OPTs connected to this network, the end user could input the received voucher code from the receipt. The code would be verified against the database, and if deemed valid, the previously unused amount is added as additional credit towards the new transaction. This flow demonstrated in the figure 3.3.

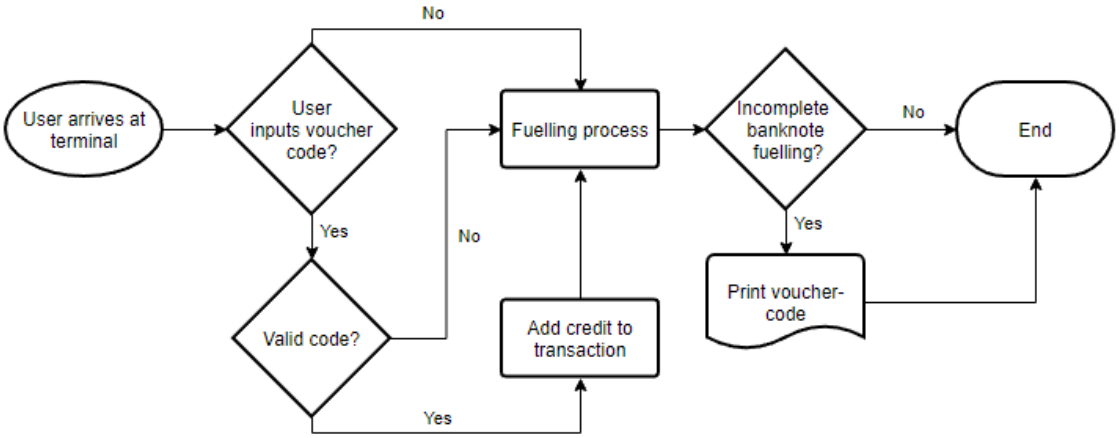

Figure 3.3. *Demonstration of the flow for voucher-based solution*

The loyalty card-based solution planned was similar to the transponder system described by Wilson [6]. The end users could be identified by already existing loyalty or discount cards or even brand-new identifying cards could be issued. The end users could identify themselves either at the beginning of the transaction or be requested an identification at the end of an incomplete banknote transaction. Identically to the voucher-based solution described above, information about the incomplete transaction would be transferred to a central database. To redeem this credit, the user could identify at any terminal for the next transaction by swiping the same card. The terminal then checks if any unused credit linked to the card is located from the central database. If found, the end user is asked if they would like to use it during the current transaction. If the end user accepts, the amount would be added to the credit of the starting transaction.

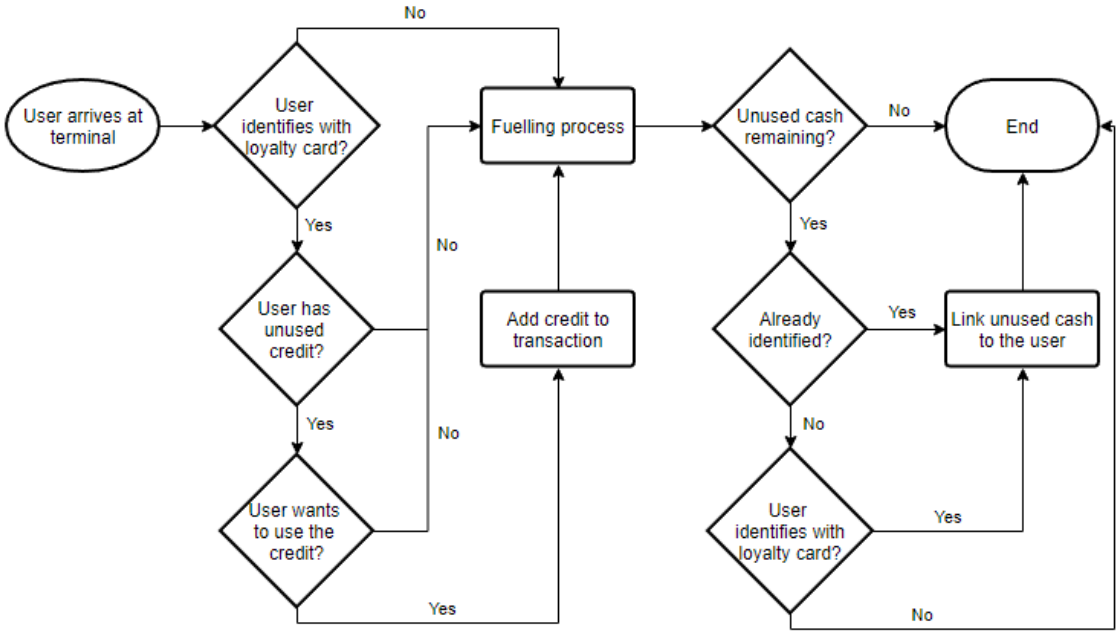

Figure 3.4. *Demonstration of the flow for loyalty card-based solution*

The flow of the loyalty-card based solution is demonstrated in the figure 3.4.

3.3.1 Selection

Selecting between the three methods of approach first came down to selecting between the mostly hardware-based solution and the other two solutions more focused on software development. The loyalty card and voucher-based solutions could be made to function with the existing terminal hardware, whereas the hardware-based solution would require installation of new hardware for the OPTs to be able to hand out any change. In addition to hardware for handing out change, the terminal would require a new coin intake for payment in order to maintain a stable supply of change independently, as stated by Ramsey and Williams [2]. Unlike the system described by Gong [3], the customer's system could not function without the use of coins. This is because the required amounts of change on a regular fuelling could not be accurately paid by using only Euro banknotes due to their large values.

As the customer already has the methods of simultaneously updating the software of many terminals over the network, the complicated process of installing new hardware seemed excessive. For the first described solution, each terminal would have to be installed with new complex hardware, which would be very time-consuming compared to a solution with only software updates or minimal hardware upgrades. For future updates and improvements, the hardware installation process could possibly even have to be repeated. Furthermore, the cash transactions in customer's terminal network are on the decline compared to new and more modern payment methods, further reducing the

necessity for new complex hardware. The hardware would also eventually require maintenance, which would further increase the costs. Maintaining a stable supply of change might also require manual labour, depending on the payment methods the end users are selecting. For these reasons, the hardware-based solution was rejected, and the focus shifted on selecting between the two software-based solutions.

The loyalty card-based solution presents a new challenge of which card to use as an identifying method. As one option, the customer could issue new cards intended for identifying end users during the transactions. Existing fleet cards could also be used for this purpose. However, this would not solve the problem for customers who are just occasionally using the customer's terminals, as they most likely would not have acquired such a card. A possible option to solve this problem would be to use already existing third-party cards. The customer's terminals already accept several different third-party loyalty and discount cards, which could possibly be used as an identifying method as an addition to customer's own cards. However, using third-party cards as the identifying information for rewarding the cash returns would require agreements with the parties in question and would present heavily increased security requirements. Furthermore, this solution would still not be valid for end users who do not own any of the cards supported by the customer.

Opposite to the loyalty card-based solution, the voucher-code based solution could be implemented without identifying the customer. The voucher code handed out by the system would itself be used as the identifier for handing out the saved credit. Because of this the voucher codes could be used similarly to regular banknotes, which would solve the problems presented by the loyalty-card based solution. In addition, the customer's OPTs already have multiple input and output devices to support the voucher code functionality: The receipt printer for printing out the voucher codes, and the keypad and the barcode scanner for possible methods of inputting the voucher code. This would allow the customer to perform the update process as a software update only without the complicated process of installing any new hardware.

By comparing the different benefits of both solutions, the voucher code-based solution was chosen as the method of approach. It had the clear benefit of being able to serve every user directly at the terminal, without having to develop complicated identification systems. The voucher code-based solution would also bind the refund to the customer's terminals, encouraging the end user to return for additional fuellings.

3.4 Designing the changes

Once the general method of approach was chosen and confirmed with the customer, the actual design work could begin. The first phase was to acknowledge and understand the customer's functional requirements and wanted use cases for the new voucher code feature. The work began by planning the feature together with the customer.

3.4.1 Use cases and requirements

When discussing the use cases from the perspective of the end user, the preferred voucher code input method quickly became clear. For the end user, the most user-friendly way to input the voucher code would be to scan it at the terminal's barcode scanner. The scan would then automatically provide the end user with the amount associated with the voucher to be used with the new transaction. The voucher codes should also be readable continuously without requiring any additional inputs from the end user, like other current payment methods. For the unfinished transactions, the voucher codes would be outputted as a barcode on the transactions receipt, from where the end user could easily acquire it and scan it as a new input in a future transaction.

From the customer's perspective, a method for controlling the voucher codes was required. The customer required the option to revoke generated voucher codes if deemed necessary in case of erroneous voucher codes. An option for printing the already existing and creating entirely new voucher codes was also required, in case of printing or voucher generation failures, and for older receipts where the barcode could have faded enough to become unreadable by the scanner.

Another design decision was whether to support voucher generation when the terminal is offline and unable to access the central database. Because the customer already required a method for creating new voucher codes in case of other failures, and the network offline time was noted to be minimal on existing terminals, the feature was agreed to work only when the terminal is online and the central database can be accessed. For offline cases, the customer would have to request change from the customer service, who could then issue an adequate voucher code to send to the end user via email, for example. This also simplified the design requirements, as the voucher codes could be controlled from a centralized location, instead of needing to synchronize the voucher data between different stations.

The customer also recognized the need for a configuration of the minimum unused amount of cash where a voucher code would be issued. This was due to the accuracy limitations on different pump devices. This minimum amount for fuelling also exists for other payment methods. For the same reason, a single voucher code could be used only once. If there would still be unused cash or voucher credit remaining after a voucher code transaction, an entirely new voucher code would be issued. The customer also required an expiry date for the generated vouchers. In summary, all this resulted in the following list of requirements for the new feature:

- Vouchers codes should be readable all the time like other payment inputs.
- A voucher code should be usable just once and have an expiry date.
- There should be a minimum amount of unused cash for a voucher to be generated.
- Feature is only required to work while online.

- In case of offline transactions and other failures, customer should be able to generate vouchers and deliver them to end users.
- In case of erroneous voucher codes, customer should be able to revoke them.

In addition to the functional requirements listed above, customer wanted to be sure that voucher codes could not be forged and could always be validated as genuine with reasonable certainty. This was to make sure that no one would be able to generate and format valid vouchers codes the real end users had yet to redeem. These requirements also had to be considered during the following design phases.

After the system requirements were settled on, the designing process continued to select the most suitable barcode symbology to use as the voucher code format.

3.4.2 Selecting the voucher code format

One initial option for the contents of the voucher code was a systematically constructed string. The string could consist of information identifying the station, terminal and transaction, with the possibility of added random data to make the code less obvious. This would exclude the possibility of any voucher code collisions, as all this information could not match for any other transaction. However, having constant information in the voucher codes could increase the possibility of users figuring out the pattern of the constructed strings. This could attract any illicit entities to attempt to get free refuellings by randomly trying to generate voucher codes, or possibly surveying stations to figure out actual codes received by customers. Any possible enticements for unwanted behaviour were deemed unacceptable, and the idea of a systematic string construct was quickly abandoned.

Another option for the voucher code was the Globally Unique Identifier (GUID). GUID is a 128-bit integer, often represented in 32 hexadecimal digits. GUIDs are ubiquitously used and due to their size and randomness, it is incredibly unlikely to get a collision when using the GUIDs as database identifiers [50]. GUIDs have 122 bits of randomness [50], making the total number of unique GUIDs 2^{122}. This makes any unauthorized attempts to generate already valid voucher codes very unlikely to succeed. Furthermore, as the GUIDs format is mostly random, no transaction information (e.g. terminal information, amount of credit) would be stored in the identifier itself, making GUID a suitable format for the content of the voucher codes.

As GUID was initially decided as the contents of the barcodes, the next step was to verify that the terminal's barcode functionalities were able to process it. A barcode symbology suitable for representing a GUID and supported by the terminal was required. As mentioned in section 3.1.1, the terminal's barcode scanner was capable of scanning most available barcode symbologies, so it would not be the initial focus on the limitations. However, the printer only supported two barcode symbologies capable of representing all the required characters in the required length of a GUID, Code 128 and QR Code. Both were supported by the barcode scanner, so the focus shifted on selecting between the

two symbologies.

First, a selection between individual character sets and formats was to be made. As GUIDs contain both lowercase letters and numeric values in a randomized order, the selection for both symbologies was clear: Based on the research conducted in sections 2.3.1 and 2.3.2, Code 128 barcode should be encoded using the character set 128B, and QR Codes should be encoding using 8-bit binary data. Both formats were tested in practice, and both symbologies were deemed functional with the GUID format. However, due to width of the receipt paper, the Code 128 symbology had to be tightly packed and scaled, with the absolute maximum length being 33 characters. The QR Code symbol fit in easily as version 3-Q, which allowed for 32 8-bit symbols with a 25% error correction capability [34].

Even though both symbologies were functional, the QR Code seemed the better option due to its error correction capabilities. As described in 3.1.1, the terminal prints receipts using a thermal printer. Because of how thermal paper works, the printed receipts tend to fade out as time progresses. This made the error correction a consequential benefit in favour of the QR Code symbology. After practical testing, the QR Code symbols also seemed to be easier to scan than the tightly packed Code 128 barcodes. After some further testing, the QR Code symbol was increased to version 4-H, which supported 34 8-bit characters with the highest error correction level of 30% [34], while still easily fitting on the receipt paper.

Once the voucher code format was chosen, it was time to design the updates to the current software components controlling the terminal.

3.4.3 OPT-Sequencer and EPS-Component

Designing the updates for the current software components started by dividing the responsibilities for handling the new functionality between the OPT-Sequencer and the EPS-Component. To keep changes to each component as manageable as possible, the feature was divided into two main functionalities: Controlling the devices and payment sequences and communicating with the centralized voucher database. This division also suited the existing architecture well, as OPT-Sequencer was already responsible for the payment terminal functionalities and communicated with the EPS-component to validate payment methods. Therefore, OPT-Sequencer was designated to control the barcode scanner and the printer to create the QR Code symbols, as well as controlling the payment sequence. The existing communication API between the OPT-Sequencer and EPS-Component would be modified to allow OPT-Sequencer to send voucher requests to the EPS-Component. EPS-Component would handle the communications to the centralized voucher database and return the generated voucher codes or redeemed credit amounts to the OPT-Sequencer as responses to the requests.

Based on the role division described above, OPT-Sequencer could use the voucher codes

similarly to any other payment method handled with the EPS-Component. However, this raised another design question, as the current OPT implementation only allows a single payment method for each transaction. This would need to be changed for voucher codes, as the credit received by an individual voucher code was expected to be quite small, with approximately 78% of the incomplete fuellings having less than 5 euros of unused cash remaining, as stated in section 3.1.4. As voucher codes only originated from cash payments, the decision was made that voucher codes would be equated as cash payments and processed as a single banknote input, and could be combined with actual banknotes to provide end users with a more fluent refuelling experience. This solution was also approved by the customer's accounting department.

Not allowing any other payment methods with voucher codes also simplified the possible problem of having unused credit left after using a voucher code in the transaction. As stated in section 3.4.1, voucher codes are redeemed in their entirety and are usable only once. This means that a new voucher code would have to be generated to refund the end user the unused cash of the new transaction. As both payment methods are equated as cash payments, no decision between the priority of the used payment methods is required, and a new voucher code can be generated for the entire unused amount. This would not be possible with e.g. a credit card payment, where the unused amount of both the voucher code and credit card payment would have to be determined to finalize the credit card payment with the correct amount.

OPT-Sequencer was also designed to maintain a record of the number of continuous and consecutive unsuccessful voucher code redemption attempts. If too many unsuccessful attempts would occur consecutively in a determined time period, OPT-Sequencer should deactivate the barcode scanner for a predetermined time period as a precaution against counterfeit barcode attempts. This is to even further ensure that no one would be able to generate a valid voucher code by illicit means.

The EPS-Component updates were designed to function very similarly to the current implementation described in section 3.1.3. During the pre-authorization (i.e. redemption) phase of the voucher code, the EPS-Component would verify the determined payment rules and discounts, and then communicate with the voucher database to validate the voucher code, returning the redeemed amount to the OPT-Sequencer. For the finalization phase, EPS-component would request the database for a voucher code generated for the amount received from OPT and return the generated code to the OPT-Sequencer. The division of the component responsibilities is illustrated in figure 3.5.

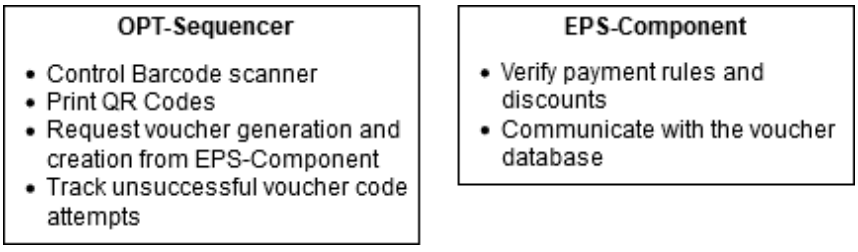

Figure 3.5. *The division of responsibilities for OPT-Sequencer and EPS-Component*

Any further design for EPS-Component's communications with the centralized voucher database was dependant on the voucher database's design and implementation, so the design focus shifted onto the new component.

3.4.4 Voucher-application

A pure database would not be enough for controlling and maintaining the voucher codes by itself. Because the database required an easy and a secure way to control it from multiple sources at once, a new program was designed to wrap the control of the database. This program, from here on called Voucher-application, would act as the interface between the database and its users.

The first design phase of Voucher-application was to divide the functionality in separate required actions. The terminals required the functionality to create and redeem voucher codes. In addition, the customer required a method for manually revoking and creating voucher codes as described in section 3.4.1. The customers manual voucher creation process and the automated process by the terminals could be seen as equal actions, which left the Voucher-application with three main functionalities: Create, Redeem and Revoke.

To allow the customer to easily navigate and search the vouchers generated by the terminals, the create-functionality was decided to require additional information about the transaction to be saved to the database. The information would include a station identifier, a terminal identifier and a transaction identifier, among others. For the use case of customer manually generating voucher codes, the fields were to be filled with predetermined constant values. The creation request would be responded to with a creation response, containing the generated code and its expiry date.

The redeem-functionality was also decided to require additional information about the station and terminal in which the code was scanned at. A redemption request would be responded to with a redemption response, containing the voucher's code, redemption status and the redeemed amount if successfully redeemed. Lastly, the revoke-functionality was decided to include information about the revoker and the reason why the code was revoked. The response would contain the voucher code and the new status of that code.

Once the functionality was designed, it was time to select the implementation architecture. After initial research, the Voucher-application was decided to be implemented as a RESTful Web service. Based on the research conducted in section 2.4, a REST web API was a good choice due to the stateless nature of the required functionality, and the natural client-server model of the EPS-component and Voucher-application. The EPS-Component would always act as the client and Voucher-application as the server. However, as RESTful APIs provide no UI, an additional program utilizing the Voucher-application would be required to provide the management UI for manual control by the customer. However, its design and implementation were left out of the scope of this

thesis.

Next step was to decide where the REST web API would be hosted. The selection was between hosting the API on-premise in the customers network or using a cloud service provider. Public cloud service providers offer some benefits compared to an on-premise solution. Most cloud applications provide an easy access from any devices, compared to the complex on-premises environments with multiple layers of internal security required to gain access [51]. Cloud-based solutions also substantially reduce hardware, software, staff and utilities costs [51]. However, if the application could make use of an already existing shared on-premise server, no additional costs would be incurred [52]. As the customer already had an internal network with the required computing resources to spare, the API was decided to be hosted on-premise on one of the servers. This also meant that most of the security concerns and firewall configurations were already taken care of due to the other related applications running in the network. The new API would run on IIS (section 2.6), already present on the server hosting other APIs. However, the customer also wanted the API to be developed in a way that it could possibly be moved to a public cloud environment in the future. This meant that some additional security options had to be addressed.

Possibly using a public cloud service provider in the future meant that information would have to be transferred via public networks. This meant that the data exchanged would have to be encrypted. The source of the requests received by the API also had to be verified, to prevent any illicit entities from accessing the API. In addition to using firewalls and IP restrictions, an X.509 certificate-based public-key cryptography was to be implemented. The certificates were discussed in section 2.5.2. The API server would have a list of trusted certificates, each dedicated to a group of fuelling stations. The API server would have its own certificate trusted by the systems at the fuelling stations. In the case of possible security issues, station certificates could be removed from the list of trusted certificates of the server, in order to prevent illicit access to the Voucher-application. New certificates could then be issued and delivered to the stations using the existing management tools for the terminal network. Initially, the system would use self-signed certificates in the closed on-premise network, but was to be implemented in a way that would enable a full PKI integration in the future, as described in section 2.5.1.

3.5 Final design

Once all the separate components were designed, the initial system design was now finalized. The following sequence diagrams are provided to summarize the functionalities of creating and redeeming voucher codes, and they can be compared to the figure 3.2, of which the unmodified functionality can be inspected from.

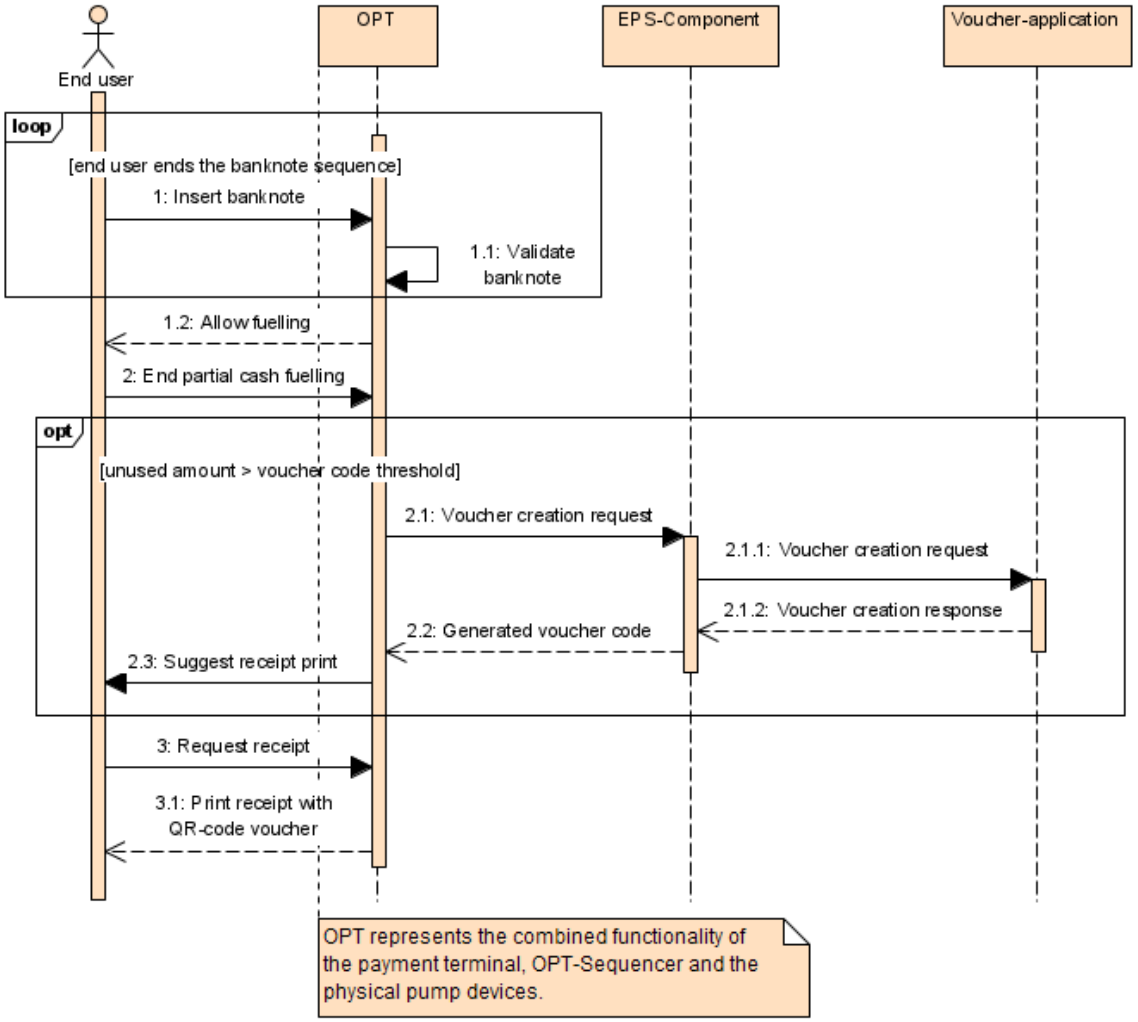

Figure 3.6. *A sequence diagram of the voucher creation sequence*

The voucher creation sequence is represented in the figure 3.6. The end user ends a cash/voucher code fuelling with a partial amount, after which the OPT-Sequencer sends a voucher creation request to the EPS-Component, which connects to the Voucher-application requesting a new voucher code to be generated. The generated voucher code is then returned to the OPT-Sequencer, which prints it to the customers receipt in the QR-Code format. If voucher code generation would fail for some reason, OPT-Sequencer would instead print a receipt with a message instructing the end user to contact the customer service for refunds.

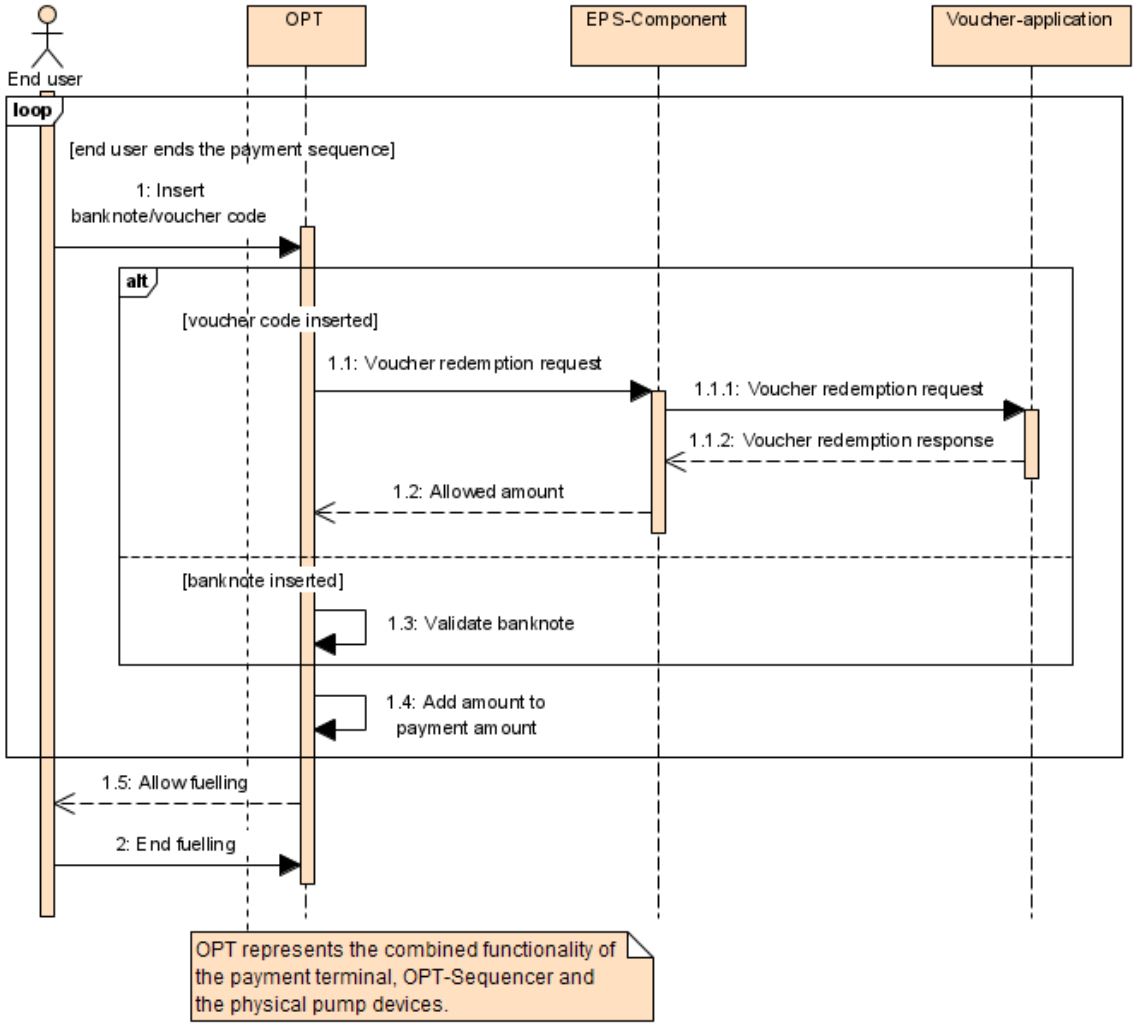

Figure 3.7. *A sequence diagram of the voucher redemption sequence*

Figure 3.7 represents the voucher redemption sequence. The end user first is in a loop of inserting banknotes or voucher codes. For every voucher code inserted, a redemption request is sent to the Voucher-application via the EPS-Component. The monetary amount of the voucher code is then added to the end user's total prepaid fuelling amount. Once the end user decides to end the payment sequence, the fuelling process is started. This fuelling can also end as a partial fuelling, which then generates another voucher-creation sequence portrayed in figure 3.6.

4 IMPLEMENTING THE CHANGES

This chapter contains a more detailed description of the implementation process of the new features. This includes the decisions made on which frameworks and programming languages to use, and descriptions about what challenges rose during the implementation. This chapter also describes the final implementations of the designed software components.

4.1 OPT-Sequencer

To better understand the implementation of the new features to the OPT-Sequencer, the in-depth understanding of the existing banknote functionality is required. The existing banknote sequence is started when the user inserts a banknote to the payment terminal's BNA. The BNA validates the inserted banknote. If the banknote is not valid, the BNA will reject it, and the sequence will not be started. However, if the banknote is valid, the terminal emits an event of an inserted banknote which can be subscribed to via the SDK described shortly in chapter 3.1.1. OPT-Sequencer is listening to the event, and the banknote sequence is started after the event is received.

The banknote payment sequence starts by confirming the banknote is acceptable according to the configured payment rules. OPT-Sequencer then directs the payment terminal to either capture or eject the banknote based on the rules. Following this, OPT-Sequencer disables all other payment input peripherals from the terminal, except for the BNA, and directs the terminal to draw the banknote insertion screen to inform the end user that the sequence was started. The value of the inserted banknote is visible in the screen. OPT-Sequencer then enters a loop of waiting for more banknote events, which lasts until the user has indicated that all the banknotes have been inserted. In this loop, the inserted banknotes are validated according to the payment rules and added to the total amount of the transaction, which is then updated to the display of the payment terminal.

After the user has indicated that all the banknotes are inserted, OPT-Sequencer will instruct the pumps to activate and allow fuelling for the inserted banknote amount. OPT-Sequencer directs the payment terminal display to inform the user that the payment process has been completed and the fuelling can begin. This entire sequence is demonstrated in detail in the figure 4.1.

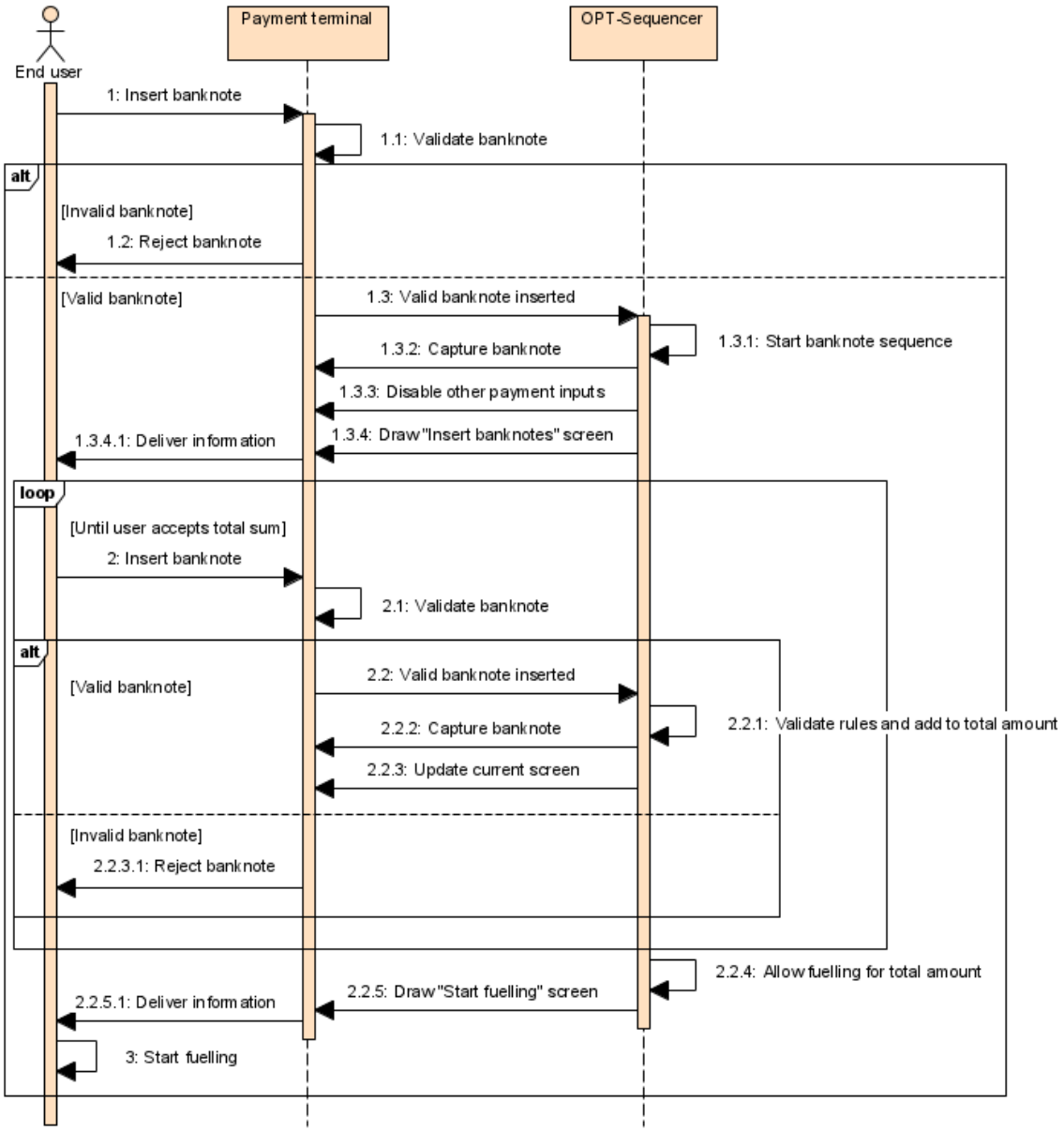

Figure 4.1. *A sequence diagram detailing the existing banknote payment sequence*

The sequence above required multiple changes to be able to support the designed voucher code functionality. Because the new features would just be extending and altering the existing OPT-Sequencer program, the framework and the language for implementation were predetermined as .NET Framework and C#-based Windows Service.

The first changes to the sequence require it to be started when the user would scan a voucher code instead of only inserting banknotes. As the sequence should work according to the designs made in section 3.5, the scanner would also have to be used during the loop of inserting further payment inputs. This required the use of the terminal's barcode scanner in order to active it and receive scan results. The terminal's SDK contained functions for controlling the scanner, but calling the native C++ functions from the OPT-Sequencer had not yet been implemented. The implementation process is described in the following section. OPT-Sequencer also required changes to the communication API

with the EPS-Component, which was implemented by simply adding additional fields to the existing API.

4.1.1 Controlling the native C++ SDK

As mentioned in section 3.1.1, the terminal SDK was in the form of a statically linked native C++ libraries. Static C++ libraries cannot be used directly from a program written in C#. Instead, they require another C++ program from which they can be called from. Native C++ code is also unmanaged, so in order to use it in a .NET solution, interoperability was required. This required the function parameters and return values to be marshaled between the managed and the native library [53]. As researched in section 2.1.3, the C++/CLI-language was the solution for this problem. The implementation of a C++/CLI library for wrapping the static library controlling the barcode scanner was required. The implemented library could then be used by the OPT-Sequencer to control the scanner.

The first step to implement this library was to create a C++/CLI project, using the static SDK libraries during compilation. The native library could then be accessed by simply including the library's header files in the calling source code. The implemented C++/CLI library would forward calls to the SDK methods and subscribe to the events emitted by the SDK when requested and return the events to the caller. The events would have to be stored in the managed memory of the OPT-Sequencer, so the data contained could be processed. To enable this, the implemented library was required to marshal the function call parameters and returned events. For most function call parameters, the runtime's default marshaling rules [53] were correct, however strings and some arrays had to be manually marshaled. Events returned by the SDK had to be manually marshaled as well, because they were formatted as custom data structures. This was accomplished by creating similar data structures in the managed C# code. The C++/CLI library would construct the new managed types based on the data in the original unmanaged event and return them to the caller instead.

Once the C++/CLI library had been implemented, its functions could be called directly from C# code. This provided OPT-Sequencer with control over the barcode scanner's functionality.

4.1.2 Printing QR Codes

OPT-Sequencer also required implementation to support the printing of QR-codes by using the terminal's printer device. The device could be controlled via the terminal SDK, similarly to the barcode scanner. As the printer was already in use, the printer functionality had already been implemented for OPT-Sequencer in earlier parts of the project, and controlling the printer required no further development.

However, a problem arose while testing the SDK's barcode printing instructions for the printer device. Other barcode symbologies worked, but QR codes could not be printed on the paper. After further investigating and contacting the SDK's developers, it was noted that the QR code functionalities for the used printer model had not yet been implemented in the current version of the SDK. The developers offered two alternate solutions: Generate the QR code symbol manually and print it as a regular bitmap image, or directly access the printer's functionalities through a pass-through command in the SDK's printer API.

After initially reviewing the two options, both were found to be functional and easily implemented. The pass-through command was chosen as the implementation method, partially due to it not requiring the use of additional third-party libraries.

4.1.3 Implementation of the new sequence

Once the required changes to support new device functionalities had been implemented and tested to be working, all that remained was integrating them to the existing sequence. A scanned QR code was assigned as an additional starting condition for the banknote payment sequence. This meant that the banknote sequence would also start when the user scanned a barcode, in addition to an inserted banknote. When enabled, the barcode scanner reads all supported barcode symbologies, which meant that the scanned code would first have to be validated to be a QR code in the right format. After this, the code would be sent to the EPS-Component for verification in the new extended field of the regular API message. If the barcode was in the wrong format, or the voucher code contained in it was not valid, OPT-Sequencer would start to track the number of unsuccessful attempts. If too many unsuccessful attempts occurred consecutively in quick succession, the voucher code functionality would be disabled by disabling the barcode scanner for a configurable time period. If the scan was instead a valid voucher code, OPT-Sequencer disables all other input methods except for the scanner and the BNA, draws the screen for inserting further payment methods, and enters the loop waiting for further payment inputs.

The entered loop functions like the one described earlier in section 4.1 and figure 4.1, with the added option of user being able to scan barcodes in addition to inserting banknotes. All the scanned codes go through the same process as the first scan, and the same restrictions for unsuccessful scans apply. Once the user exits the loop, the fuelling process would begin. A sequence started by a barcode scan is demonstrated in figure 4.2.

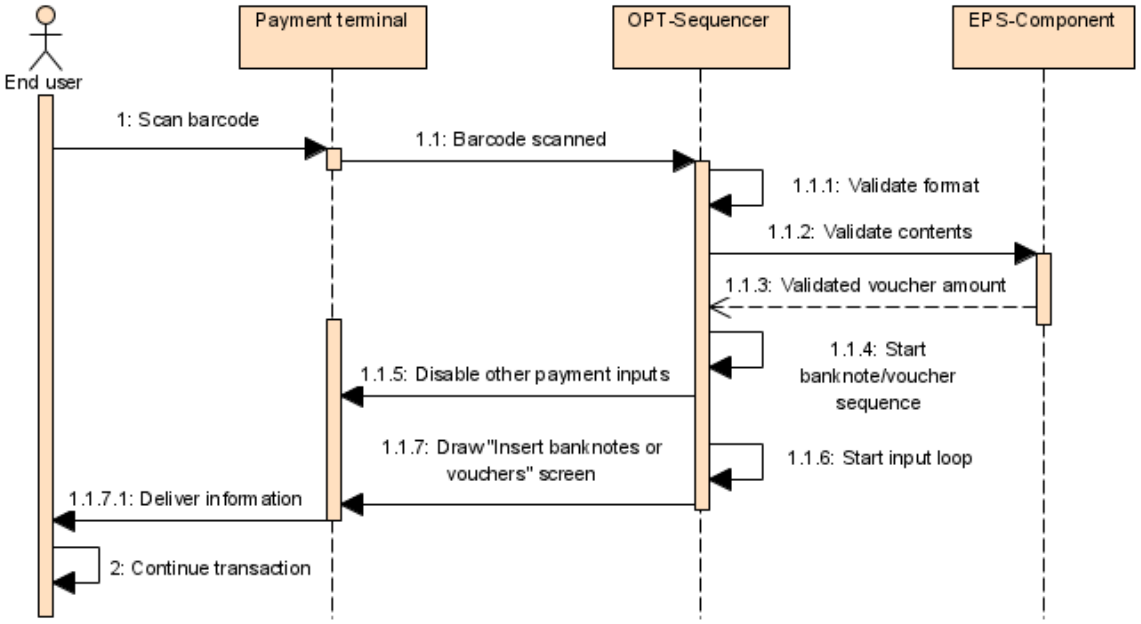

Figure 4.2. *A sequence diagram detailing the start of the sequence by scanning a barcode*

After the fuelling is completed, OPT-Sequencer must now check for unused amounts. In the prior implementation, any transactions using banknotes required no further action. The transaction was marked as completed and the receipt was generated. This required a change. Should any unused money remain at the end of a banknote/voucher transaction, the transaction would be directed to a similar finalizing process as with electronic payment methods. The finalizing process would use another extended field of the EPS-Component API to request a voucher code to be generated for the remaining amount. Should the voucher code generation succeed, OPT-Sequencer would generate the receipt containing the voucher code. Terminal's display would also be used to suggest the user to print the receipt for the transaction. Once the user had decided to print the receipt, OPT-Sequencer would use the newly adapted pass-through functions of the printer device to print the receipt with the corresponding QR Code. Should the voucher code generation fail for any reason, OPT-Sequencer would replace the QR code in the receipt with a suggestion to call the customer service to get a refund for the lost credit. Similar instructions would be drawn on the terminal's display.

4.2 EPS-component

The implementation changes required for the EPS-Component were minor. The first change was to implement the new extension fields to the API used to communicate with OPT-Sequencer. After the new fields were added, EPS-Component had to separate the voucher code requests from regular payment requests in order to process them correctly. The selection was made based on the contents of multiple fields in the API requests.

To keep old functionality unchanged, the voucher requests were directed to a different dedicated sequence.

Like the existing sequence described in section 3.1.3, the voucher sequence would validate any existing payment and product rules, and then proceed on to call the REST API of the Voucher-application. However, as the API was designed to use X.509 certificates for additional security, implementing the client-side of the API required some additional work. The computer running EPS-Component would be assigned a password protected private key file, which would be used by the EPS-Component to securely send requests to the Voucher-application API. The private key password and file were stored in configurable locations, as it is impossible to store them securely within the software [54]. To validate the responses from the api, the public X.509-certificate of the Voucher-application would be added to the computer's list of trusted certificates.

After the certificate-functionality was implemented, the actual API-requests were implemented. The details of the API are further disclosed in section 4.3.2. After the client-side of the REST API was implemented and tested to function properly, the responses were set to be returned to the OPT-Sequencer in the extended fields of the API-messages implemented in the beginning of this section.

4.3 Voucher-application

As the Voucher-application was an entirely new component, more freedom was provided for the implementation framework and techniques. The initial options for the framework were ASP.NET and ASP.NET Core based on .NET Framework and .NET Core, respectively. Main reasons for the selection were as follows: All the other software components in the project had been implemented using .NET, which had led to a familiarity with the .NET environment and its database implementations. The design of the Voucher-application was a simple REST API with database connection. Both options contained a built-in application model for developing REST APIs [55], which would help the implementation of the API. As stated in section 2.6, both were also supported by IIS, which was the initial selection for the web server hosting the new service.

The next step was to select the best option between the two frameworks. Regarding the possible future migration to cloud environments, both frameworks are supported by multiple cloud service providers, including Google Cloud [56], Amazon Web Services [57] and Microsoft Azure [58]. However, ASP.NET Core was built with cross-platform support and it supported Windows, macOS and Linux compared to ASP.NET, which only supported Windows environments [59]. This enabled multiple additional ways to host ASP.NET Core services compared to ASP.NET services. ASP.NET Core also had a cloud-ready environment-based configuration system, and improved performance compared to ASP.NET [59]. It was also optimized for both cloud and on-premises applications [10], making it ideal for the use-case in question. The Voucher-application was to be im-

plemented using ASP.NET Core framework.

ASP.NET Core supports the programming languages C# and F#. Both languages were suited for implementing REST APIs. F# usually has more performance compared to C# when doing computations [60], but based on the Voucher-application design, heavy computation would not be required from the service. C# was a familiar language, and it was already used in every other software component of this project. Based on this, C# was selected as the programming language for Voucher-application.

4.3.1 Structure

Implementing the Voucher-application started by initiating a built-in ASP.NET Core Web API application template matching the Web API application model. This provided a base program any ASP.NET Core Web API could be built on. The template provides the starting points for the program, and an initial empty endpoint for HTTP requests. In the ASP.NET Core Web API template the endpoints for REST APIs are created by deriving a custom controller class from the class ControllerBase, which provides multiple properties and methods useful for handling HTTP requests [61]. A VoucherController-class was created to function as the endpoint for the HTTP requests.

It is good practice to keep business logic out of controllers, and instead delegate the functionality to different services within the program [10]. Therefore, a separate service class containing the required functionality called VoucherService was created. To allow future expandability, the class would implement an interface. This interface would contain the methods required to provide the required functionality for the controller to implement the API. The controller would then call the methods of the implemented interface to access the required functionality. The created service implementing the interface was added to the program using the optional built-in method ConfigureServices, which uses dependency injection to make the created service available in the program [62] by adding them to a service container. Dependency injection is a software design pattern, which is a technique for achieving Inversion of Control between classes and their dependencies [63]. Using dependency injection improves code maintainability and application testing, increases code reusing and reduces class coupling [64].

The implemented service would require access to the voucher-database to control it. Database access was implemented using Entity Framework (EF) Core. EF Core enabled developers to use .NET objects when working with a database, eliminating the need to write most of the data-access code usually required [65]. EF Core supports multiple different databases [66], which made it a suitable solution considering the possible cloud migration in the future. EF Core works with the use of a model. A model is made of entity classes representing database tables and a context object representing a session with the database [65]. Implementation continued by creating the EF core model of the voucher-database. EF core supports providing the database connection context using

dependency injections [67]. To ease testing and allow future expandability, the context would be provided using dependency injection during the startup of the service, based on different configuration settings. The structure of the Voucher-application is represented in figure 4.3.

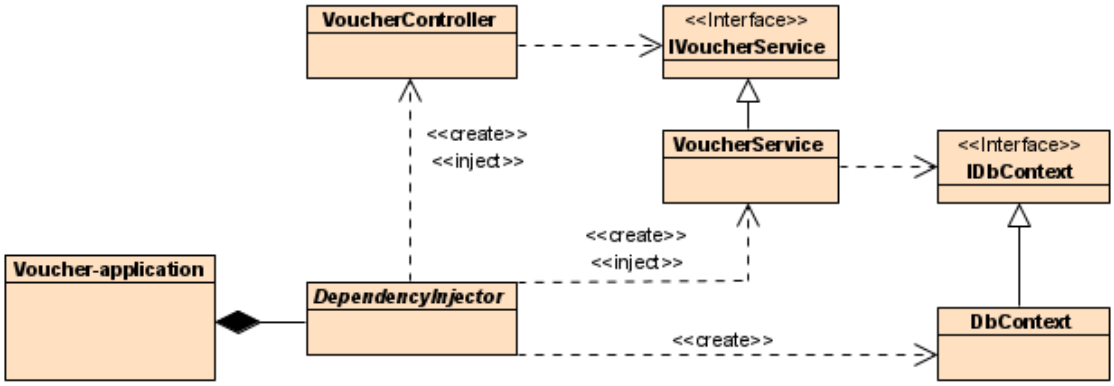

Figure 4.3. *A class diagram representing Voucher-application structure*

In the class diagram, the abstract DependencyInjector-class represents the built-in ASP.NET Core implementation for dependency injections. It handles the creation and injection of the implementations for the required interfaces, which helps at keeping the code manageable.

To enable the security designed in section 3.4.4, HTTPS was enabled for the Voucher-application. Voucher-application was set to require certificates from all requests to the API and validate the certificates against the list of trusted certificates issued to stations. Voucher-application was assigned a server certificate in order to enable clients to verify the responses.

To even further ease the testing and implementation process, an OpenAPI implementation was added to the Voucher-application. OpenAPI is a "language-agnostic specification for describing REST APIs" [68]. The OpenAPI implementation can be used to generate the OpenAPI specification of the implemented REST API. The specification can be used to generate interactive documentation, client SDK generation and API discovery [68].

4.3.2 Finalized API

The final part of the implementation was the REST API. The first step was to create the models. A model is a class representation of the data managed by the application [69]. One request model and one response model were created for each API method. As designed in section 3.4.4, the initial methods for the API were Create, Redeem and Revoke.

After further investigation to the design, a new method was created. The initial Redeem-method was designed to validate the voucher-code and then mark it as redeemed. This was implemented as two separate tasks, validation phase and the redemption phase. Because the customer might want to validate the authenticity of voucher codes without redeeming them, Validate was added as a new API method. This method could be used to validate voucher codes without marking them as redeemed.

The responses from these methods were implemented as identical, so both Redeem and Validate could use the same model for response. The same design was then implemented to Revoke as well, because its response could also be represented using the same response model. The final implemented REST API methods and models are represented in figures 4.4, 4.5, 4.6 and 4.7.

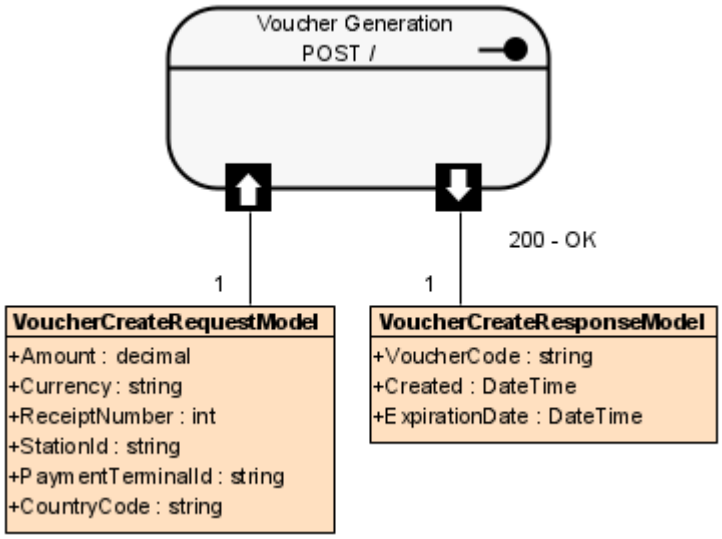

Figure 4.4. *Voucher generation class diagram*

Generating voucher codes required information about the transaction and the remaining amount. The StationId, PaymentTerminalId and ReceiptNumber attributes enable the identification of the terminal and transaction, whereas Amount represents the remaining unused amount. In addition to the required information, the attributes Currency and CountryCode were added as currently optional attributes. This was in order to enable possible future support for multiple countries, should the same database be used for different countries.

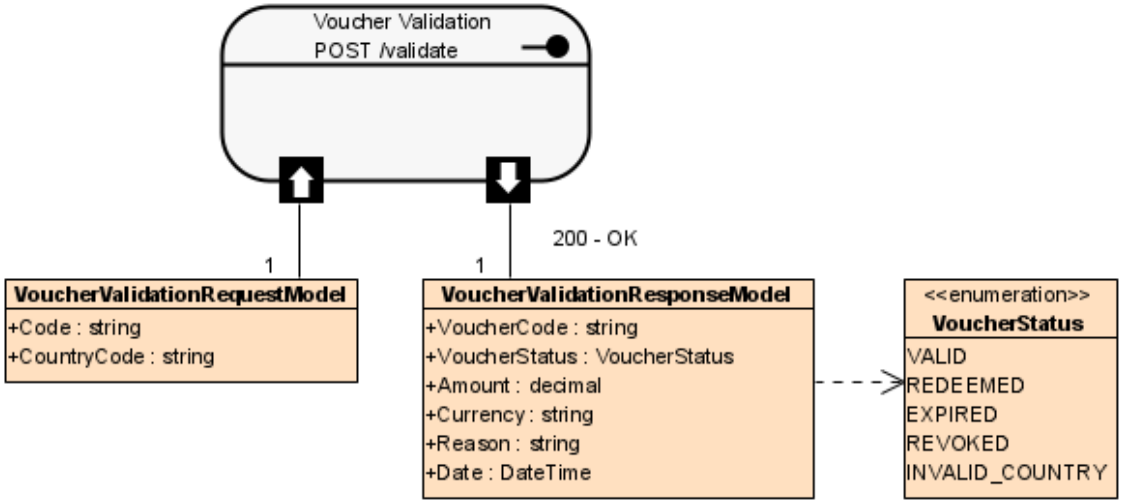

Figure 4.5. *Voucher validation class diagram*

For validation, only the voucher code is required from the request. The response model contains the voucher code and amount, as well as additional information of the voucher. VoucherStatus indicates the voucher's current status as an enumeration value. Date is a conditional attribute representing the time when either voucher redemption, expiration or revocation occurred. Reason is a return value stating the reason for the voucher's possible revocation. It is empty if the voucher has not been revoked.

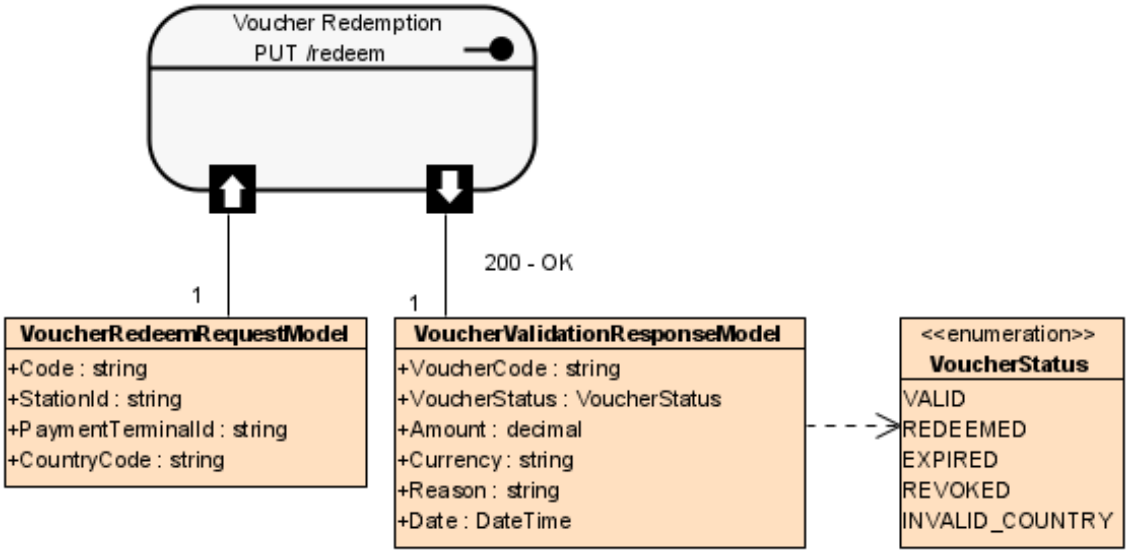

Figure 4.6. *Voucher redemption class diagram*

In addition to the voucher code, redemption requires information about the terminal where the request is originating from. The response model is identical to the one used in validation, representing the new status of the voucher.

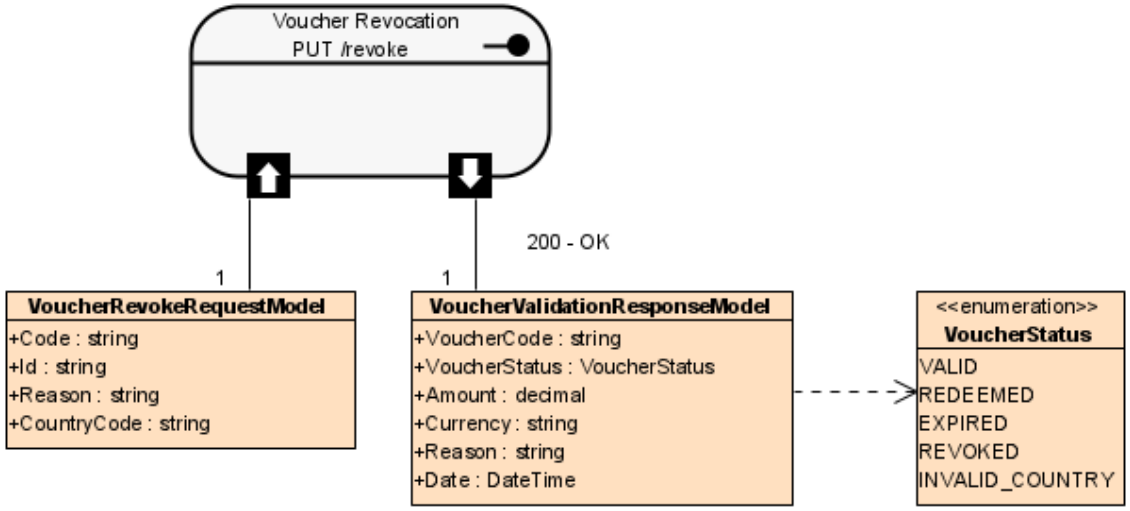

Figure 4.7. *Voucher revocation class diagram*

In voucher revocation, the required attributes are the voucher code, an identifier of the revoker and the reason why the code is being revoked. The response model is the same model as for voucher validation. Failed requests would be responded to either with status codes 404 for voucher codes not found from the database or 500 for internal server errors, such as database connectivity issues.

After the REST API had been implemented and its functionality tested, the Voucher-application was finished. The functionality was mainly tested using unit tests and the testing tools provided by the OpenAPI implementation. The security of the Voucher-application was not comprehensively tested at this point, beyond the cryptography aspects. This was because the application was initially to be hosted on the customer's internal terminal network. Unauthorized access to this network should not be possible, and the access itself could prove more harmful than the compromising of the Voucher-application. The comprehensive security testing will become important at a later part of the project. As Voucher-application was the last software component to be developed, the entire solution was now ready for integration testing in a Quality Assurance (QA) environment before updating it to production.

The integration testing was performed by using the QA environment for testing the entire pipeline of the developed solution. The testing was performed for both successful and unsuccessful cases. The test results were promising, and the solution seemed to work as intended. The solution was deemed ready to be rolled out to production to improve the existing terminal network.

5 EVALUATION

This chapter contains the evaluation of the implemented functionality and further investigation to the results produced. It seeks to answer the research questions established for the thesis and reflects on the possible future extensions and development of the created functionality.

5.1 Resulting impacts

By the time of writing this thesis, the software had not yet been completely adapted to production. It had been in use at two separate stations as a pilot version of the software. The pilot had been ongoing for approximately eight months, but due to reasons unrelated to this thesis, it had not yet been updated to more stations. Therefore, a direct comparison to the data based on the interrupted banknote transactions of the whole terminal network could not be produced. Individual stations also experience variations on banknote usage between short periods of time due to the different users of the terminals. This meant that direct comparison between short periods of time with and without the new features would produce very different results based on the comparison period selected. However, some conclusions could still be drawn from the usage statistics of the new features. These statistics are featured in table 5.1.

Table 5.1. *Overall usage statistics of the new functionalities over the pilot period*

Station	Vouchers generated	Average amount	Vouchers redeemed	Redemption rate
Station 1	13	11.40 €	7	53.85%
Station 2	6	7.45 €	3	50.00%
Combined	19	10.15 €	10	52.63%

A total of 19 voucher codes had been issued by the new system, as can be seen from the table above. This resulted in approximately 4.75 hours of saved labour, based on the 15-minute handling time of each manual cash refund presented in section 3.1.4. By the time of writing, no known malfunctions or issues had been encountered while creating or redeeming the voucher codes. This meant that so far, no additional labour on cash refunds

had been required by the customer for these two stations. The average amount could not really be interpreted, as the low number of vouchers generated meant that individual transactions with high or low unused amounts could fluctuate the average greatly.

The number of vouchers generated by Station 1 was greater than that of Station 2. This is because Station 1 was the initial pilot station and the new features had been in use for longer than at Station 2. When looking at the overall usage, the redemption rate of the voucher codes was also rather low. This was most likely due to the low total number of codes generated by the system, as not all users had had the time to redeem their newest voucher codes yet, decreasing the redemption rate. The generation rate of the voucher codes also seemed to fluctuate on different months during the pilot period. In order to get a table more representative of what long-time functionality of the entire network could look like, new data was chosen from a one-month period with high voucher code generation. In this period, both stations had had the new system operational, and users had had multiple months to redeem their voucher codes. This data is represented in table 5.2.

Table 5.2. *Usage statistics during a one-month period*

Station	Vouchers generated	Average amount	Vouchers redeemed	Redemption rate
Station 1	4	5.29 €	3	75.00%
Station 2	3	4.86 €	3	100.00%
Combined	7	5.11 €	6	85.71%

Comparing these two tables reveals that each station had had relatively the same amount of generated voucher codes. This further suggested that both stations were functioning properly, and no immediate issues were present. The redemption rate of the generated voucher codes also went up, most likely due to enough time passing for users to have redeemed their codes.

The high redemption rate could suggest that the new feature was user-friendly and intuitive to use. This further suggests that the user experience of the OPT had increased due to the developed features.

However, for more comprehensive conclusions to be drawn about the system, more user data would be required. The amount of data available at the point of writing was not inclusive enough to make proper comparisons and evaluations about the performance of the developed features.

5.2 Solution and technologies

Evaluating the adequacy of the selected solution and technologies could also have used more data. However, based on the mostly fluent implementation process of the new fea-

tures and the easy update process of a software-only solution, the voucher-code based solution proved to have lots of benefits with few drawbacks. One drawback could be the fact that voucher codes are generally small in their monetary value and they can only be paired with cash or other voucher codes in order to fuel larger quantities of fuel. This could be an inconvenience to users only occasionally using cash as their payment method. However, based on the functionality of the two pilot stations and no customer service requests relating to the voucher codes, the selected solution further seemed to be a suitable fit for the problem.

Based on the limited data from the stations, the selection of the QR Code as the voucher code format also seemed successful. When reviewing older data, the redemption rate of the generated voucher was high. This would imply that the generated QR Codes were suited for the purpose. They could be scanned without problems and would not get worn out from the receipt paper before the user has had a chance to use them at the next refill.

5.3 Future of the project

The next step for the project would be to distribute the new features in a larger scale, and eventually to the entire terminal network. This would provide more data about how well the project succeeded. After more usage data has been generated and the current functionality has been evaluated in a larger scale, further updates and improvements could be implemented to the project in the future. Two of these new possible features are discussed here, along with the future challenges they may present.

5.3.1 From On-Premise to Cloud

One of the possible development tasks for the future was migrating the service from On-Premise hosting to cloud hosting. As mentioned in section 3.4.4,this was already known during the development of the current features. It had already been preliminarily planned for during the design and implementation phase, as discussed in sections 3.4.4 and 4.3. However, not everything could be accounted for during the implementation.

The obvious initial challenge would be to select a suitable cloud service provider. This would require comparing different cloud service providers and the services they provide. This includes designing the best configuration for hosting the new features. Should the new features be hosted on a single virtual machine similar to the current implementation, or should each component be hosted in services dedicated to host certain types of components, e.g. database in a database service?

The selections could not be based on the requirements of this solution alone, as it would be potentially affected by customer's other services that already have been or are planned to be migrated to a cloud environment. The selections would also be affected by the

pricing of the cloud service provider and their services.

Regardless of the selected cloud service provider and implementation style, additional development would be required to integrate to the selected cloud service provider's services. This includes further investigation and testing on how to connect from the terminal network to the Voucher-application hosted in the cloud. A steady connection to the cloud would be a requirement, as the voucher functionality was not designed to work offline. Overall, the migration would require further testing and piloting to ensure the entire solution works fluently and securely.

5.3.2 International distribution

Another idea for further development was to distribute the solution to the customer's terminal networks of other countries. This subject was touched on in section 4.3.2, where some country-specific parameters were added to the REST API. As this matter was initially not considered during the design phase, further design questions would have to be answered. Could the voucher codes be used across different countries and their dedicated terminal networks? Can the same database be used or should vouchers be contained separately for each country?

Further research would also be required on the laws of each country and how they affect requirements for systems. These requirements would have to be taken into account while distributing the system and could cause additional development to any component. Another point of attention would be the integration of the newest software additions to the terminal softwares of other countries, which have not been running the same version due to different requirements on the terminal functionalities.

Another matter of investigation would be the accounting system of each country. Could voucher codes similarly be seen as equivalent payment methods to banknotes, or is a different approach required. The systems and requirements in new countries could open the door for additional payment methods to be used alongside voucher codes, or remove other methods altogether.

6 CONCLUSION

The topic of this thesis was to solve an issue present at a customer's outdoor fuelling terminals. The terminals were unable to return the unused cash if the fuelling had been interrupted before all the cash was used. Customer's return process for this unused cash required labour from their customer service and accounting teams. This thesis aimed to answer the question of how much resources could be saved by automating the return process. Furthermore, which was the most suitable solution to solve this problem, and which were the best technologies to implement it.

The work done started by investigating the customer's current payment terminal system and its different software components. This provided understanding of the functionality of the current system and provided further insight into the problem at hand. Software components relevant to the problem were identified as the OPT-Sequencer and the EPS-Component. The payment terminal and its provided SDK was also recognized. After studying the current system, the focus shifted on researching other existing solutions for similar problems. This provided better understanding of the issue and insight on how it had been solved by others and how the problem should be approached.

The research lead to multiple existing solutions for the problem in the form of patented systems. These patents contained approaches based on different hardware solutions, software solutions, or a combination of both. After studying the patents, three different methods of approach to solve the problem were planned: A hardware-based solution, a loyalty card-based solution and a voucher-code based solution. After comparing the different methods of approach, the voucher-based solution proved to be the most suitable solution due to its easy software-only update process, because it did not require any identifying information about the users, and its low management requirements.

The voucher-based solution was to automatically issue the end user with a voucher-code in case an interrupted cash fuelling occurred. The voucher code would also be saved to a central database. The user could use this voucher code at any of the terminals in customer's terminal network to add the unused cash amount to their next fuelling transaction.

After the general idea for the solution had been selected, the work shifted to identifying further design questions. Contemplating these questions with the customer led to the determination of the use cases and requirements for the solution.

The solution was then designed with the help of literature and online sources. The design process resulted in a new software component, called the Voucher-application, which

would provide a method of controlling the voucher database via a REST API. The utilization of this new application required minor alterations to the OPT-Sequencer and EPS-Component. The optimal voucher code format was recognized as GUID due to its highly unique and random nature. The optimal representation for the voucher code proved to be the QR Code barcode symbology. This was due to the symbology's error correction capabilities and ability to store information in a small area. The symbol also fit well to the used receipt paper. The payment terminal's existing barcode scanner and printer device were also able to utilize the QR Code symbology without hardware modifications.

The next step was to implement the designed changes. The selected approach and design proved successful during the implementation phase and most problems had been accounted for during the design phase. However, the selection of QR Code symbology presented a problem while using the printer via the terminal SDK, but this was quickly solved with the help of the SDK's developers. Even though most of them were pre-determined due to the project being an extension on already existing software components, the selected technologies proved suited for the purpose. The extensions developed for the existing Windows Service software components seemed functional and succeeded well. Wrapping the API of the barcode scanner using C++/CLI also succeeded without issues. The selection of ASP.NET Core as the framework for the Voucher-application proved functional. During the design phase, the requirement for a separate validation-command was not recognized, but this was realized and fixed during implementation the phase.

At the time of writing, the resulting impacts of the implemented system were not clear. The new features had only been updated to two separate pilot stations, and more data would be required to draw any unambiguous conclusions. This meant that the question of how much resources can be saved by automating the cash return process could not be concisely answered. Only a total of 19 voucher codes had been issued, which resulted in approximately 4.75 hours of saved labour time since the start of the pilot period. The number of vouchers generated by the two stations varied, mainly because the other one had been in operation for longer. In overall statistics, the redemption rate of the vouchers was only 52.63%.

However, some results could be interpreted when reviewing a one-month period where both stations had been active, and users had had enough time to redeem their voucher codes. Formatting the data this way revealed that both stations had issued relatively same number of voucher codes, and the combined redemption rate of vouchers increased to 85.71%. This could suggest that the new features were working as intended and that the choices on technologies and voucher code formats were successful. Furthermore, the high redemption rate could suggest that the new features had the potential to be adapted by the terminal users in the long run.

After enough usage data is generated in the future to validate the functionality of the new features, additional development ideas could be pursued. One idea was a migration from On-Premise hosting to cloud hosting. This potential feature had already been considered during the design and implementation phase, and it would mostly only require work based

on the chosen cloud service provider and their provided services. Another point of focus for further development would be the distribution of the new features to terminal networks in other countries. This would still require work based on the laws and requirements of the target countries.

The main goal of this thesis was to provide an accurate description of the entire development process of the new features, which succeeded well. The thesis covered the research, design and implementation phases of the project. However, the testing of the solution was not comprehensively covered, and the eventual large-scale deployment had not yet been achieved. This thesis should still help readers better understand what to consider when designing and implementing a system with similar functionalities. Two of the three research questions established were answered with concise answers, but due to the lack of large-scale deployment, the amount of resources saved by the deployment of this project was still undetermined. Overall, the thesis succeeded well.

REFERENCES

[1] Wise Guy Reports. *Outdoor Payment Terminal (OPT) Market - Global Industry Analysis, Size, Share, Growth, Trends and Forecast 2018 - 2025*. English. July 9, 2018. URL: https://www.wiseguyreports.com/reports/2842615-global-outdoor-payment-terminal-opt-market-size-status-and-forecast-2025 (visited on 09/26/2019).

[2] F. D. Ramsey and J. Williams. Unattended automated system for selling and dispensing fluids, with change-dispensing capability. U.S. pat. 6,055,521. Nov. 13, 1998.

[3] X. Gong. Paper money delivery passage, self-service terminal capable of being equipped with multiple cash boxes and self-service fee payment change-making refueling service terminal. Pat. CN103150851. Feb. 5, 2013.

[4] G. Miller, H.-G. Jendges and N. Crynen. Vending type machine dispensing a redeemable credit voucher upon payment interrupt. U.S. pat. 5,055,657. July 28, 1989.

[5] M. Kurowski, T. P. Bruskotter and E. M. Swapp. Automated filling station with change dispenser. U.S. pat. 5,895,457. Oct. 7, 1997.

[6] A. H. Wilson. Fuel dispensing system for cash customers. U.S. pat. 7,027,890. Dec. 4, 2000.

[7] C. Adams and S. Lloyd. *Understanding PKI: Concepts, Standards, and Deployment Considerations*. Boston, MA 02116, USA: Pearson Education, Inc, 2003. ISBN: 0-672-32391-5.

[8] J. R. Vacca. *Public Key Infrastructure: Building Trusted Applications and Web Services*. Boca Raton, FL 33487-2742, USA: Taylor & Francis Group, LLC, 2004, pp. 8. ISBN: 978-0-203-49815-6.

[9] R. T. Fielding. Architectural Styles and the Design of Network-based Software Architectures. PhD thesis. University of California, Irvine, 2000.

[10] F. Reynders. *Modern API Design with ASP.NET Core 2 Building Cross-Platform Back-End Systems*. English. 1st edition. Berkeley, CA: Apress, 2018. ISBN: 1-4842-3519-3.

[11] M. Massé. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. Sebastopol, CA 95472, USA: O'Reilly Media, Inc, 2012, pp. 1–10. ISBN: 978-1-449-31050-9.

[12] Microsoft. *What is .NET?* English. URL: https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet (visited on 10/10/2019).

[13] Microsoft. *.NET architectural components*. English. Aug. 23, 2017. URL: https://docs.microsoft.com/en-us/dotnet/standard/components (visited on 10/10/2019).

[14] Microsoft. *.NET Standard*. English. Sept. 23, 2019. URL: https://docs.microsoft.com/en-us/dotnet/standard/net-standard (visited on 10/19/2020).

[15] Microsoft. *What is "managed code"?* English. June 20, 2016. URL: https://docs.microsoft.com/en-us/dotnet/standard/managed-code (visited on 10/10/2019).

[16] *Common Language Infrastructure (CLI)*. Standard ECMA-335. Ecma International, June 2012.

[17] Microsoft. *Common Language Runtime (CLR) overview*. English. Apr. 2, 2019. URL: https://docs.microsoft.com/en-us/dotnet/standard/clr (visited on 10/11/2019).

[18] *C++/CLI Language Specification*. Standard ECMA-372. Ecma International, Dec. 2005.

[19] Microsoft. *Native and .NET interoperability*. English. Nov. 4, 2016. URL: https://docs.microsoft.com/en-us/cpp/dotnet/native-and-dotnet-interoperability?view=vs-2019 (visited on 10/11/2019).

[20] Microsoft. *Overview of Marshaling in C++/CLI*. English. July 12, 2019. URL: https://docs.microsoft.com/en-us/cpp/dotnet/overview-of-marshaling-in-cpp?view=vs-2019 (visited on 10/11/2019).

[21] Microsoft. *.NET Framework versions and dependencies*. English. Apr. 18, 2019. URL: https://docs.microsoft.com/en-us/dotnet/framework/migration-guide/versions-and-dependencies (visited on 10/17/2019).

[22] Microsoft. *Overview of the .NET Framework*. English. Mar. 30, 2017. URL: https://docs.microsoft.com/en-us/dotnet/framework/get-started/overview (visited on 10/17/2019).

[23] Microsoft. *Get started with the .NET Framework*. English. Apr. 2, 2019. URL: https://docs.microsoft.com/en-us/dotnet/framework/get-started (visited on 10/17/2019).

[24] Microsoft. *Introduction to .NET*. English. Sept. 28, 2020. URL: https://docs.microsoft.com/en-us/dotnet/core/introduction (visited on 10/19/2020).

[25] Microsoft. *What's new in .NET 5*. English. Oct. 13, 2020. URL: https://docs.microsoft.com/en-us/dotnet/core/dotnet-five (visited on 10/19/2020).

[26] Microsoft. *Introduction to Windows Service Applications*. English. Mar. 30, 2017. URL: https://docs.microsoft.com/en-us/dotnet/framework/windows-services/introduction-to-windows-service-applications (visited on 05/15/2020).

[27] Microsoft. *Service Control Manager*. English. May 31, 2018. URL: https://docs.microsoft.com/en-us/windows/win32/services/service-control-manager (visited on 05/15/2020).

[28] Microsoft. *ServiceController Class*. English. URL: https://docs.microsoft.com/en-us/dotnet/api/system.serviceprocess.servicecontroller?view=netframework-4.8 (visited on 05/15/2020).

[29] *Code 128 bar code symbology specification*. Standard ISO/IEC 15417:2007. International Organization for Standardization, June 2007.

[30] N. J. Woodland and B. Silver. Classifying apparatus and method. U.S. pat. 2,612,994. Oct. 20, 1949.

[31] S. Roberts. George Laurer, Who Developed the Bar Code, Is Dead at 94. English. *The New York Times* (Dec. 11, 2019). URL: https://www.nytimes.com/2019/12/11/technology/george-laurer-dead.html (visited on 05/19/2020).

[32] G.F. Why QR codes are on the rise. English. *The Economist* (Nov. 2, 2017). URL: https://www.economist.com/the-economist-explains/2017/11/02/why-qr-codes-are-on-the-rise (visited on 05/19/2020).

[33] *QR Code bar code symbology specification*. Standard ISO/IEC 18004:2015. International Organization for Standardization, Feb. 2015.

[34] D. W. Incorporated. *Information capacity and versions of the QR Code*. English. Mar. 30, 2017. URL: https://www.qrcode.com/en/about/version.html (visited on 05/20/2020).

[35] H. W.-d. Ahmad. *Building RESTful Web Services with PHP 7*. Birmingham, UK: Packt Publishing Ltd, 2017, pp. 8–21. ISBN: 978-1-78712-774-6.

[36] Microsoft. *Public Key Infrastructure*. English. May 31, 2018. URL: https://docs.microsoft.com/en-us/windows/win32/seccertenroll/public-key-infrastructure (visited on 09/02/2020).

[37] B. Ballad, T. Ballad and E. Banks. *Access Control, Authentication, and Public Key Infrastructure*. Sudbury, MA 01776, USA: Jones & Bartlett Learning, LLC, 2011. ISBN: 978-0-7637-9128-5.

[38] R. Barnes, M. Thomsom, A. Pironti and A. Langley. *Deprecating Secure Sockets Layer Version 3.0*. RFC 7568. Internet Engineering Task Force (IETF), June 2015.

[39] Microsoft. *X.509 Public Key Certificates*. English. May 31, 2018. URL: https://docs.microsoft.com/en-us/windows/win32/seccertenroll/about-x-509-public-key-certificates (visited on 09/03/2020).

[40] Microsoft. *X.509 Public Key Certificates: Basic Fields*. English. May 31, 2018. URL: https://docs.microsoft.com/en-us/windows/win32/seccertenroll/about-basic-fields (visited on 09/03/2020).

[41] W3Techs. *Usage statistics of web servers*. English. URL: https://w3techs.com/technologies/overview/web_server (visited on 09/03/2020).

[42] Microsoft. *IIS Overview*. English. URL: https://www.iis.net/overview (visited on 09/03/2020).

[43] Microsoft. *IIS Web Server Overview*. English. Nov. 16, 2007. URL: https://docs.microsoft.com/en-us/iis/get-started/introduction-to-iis/iis-web-server-overview (visited on 09/03/2020).

[44] Microsoft. *ASP.NET and PHP Support*. English. URL: https://www.iis.net/overview/choice/aspnetandphpsupport (visited on 09/04/2020).

[45] Microsoft. *Host and deploy ASP.Net Core*. English. Feb. 7, 2020. URL: https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/?view=aspnetcore-3.1 (visited on 09/22/2020).

[46] Microsoft. *Introduction to IIS Architectures*. English. Nov. 16, 2007. URL: https://docs.microsoft.com/en-us/iis/get-started/introduction-to-iis/introduction-to-iis-architecture (visited on 09/04/2020).

[47] J. M. Enright. Cash delivery apparatus for motor fuel dispenser or other self service facility. U.S. pat. 6,845,907. Nov. 12, 1999.

[48] K. Gilbert, G. A. Welstad and T. Welstad. Automated voucher cash-out system and method. U.S. pat. 5,953,709. Feb. 19, 1998.

[49] R. A. Luciano Jr. Voucher-based terminals for use in a gaming system. U.S. pat. 6,908,384. Sept. 12, 2003.

[50] J. Singleton. *ASP.NET Core 2 High Performance*. 2nd edition. Birmingham, UK: Packt Publishing Ltd, 2017, pp. 181. ISBN: 978-1-78839-976-0.

[51] S. Fenton. The high cost and risk of On-Premise vs. Cloud. English. *CIO* (2017). ISSN: 0894-9301.

[52] P. Ayton. Cloud Vs On Premise: Which One Should Your Firm Choose? English. *Mondaq business briefing* (2018).

[53] Microsoft. *Type marshaling*. English. Jan. 18, 2019. URL: https://docs.microsoft.com/en-us/dotnet/standard/native-interop/type-marshaling (visited on 09/16/2020).

[54] Microsoft. *Handling Passwords*. English. May 31, 2018. URL: https://docs.microsoft.com/en-us/windows/win32/secbp/handling-passwords (visited on 09/18/2020).

[55] Microsoft. *ASP.NET Web APIs*. English. URL: https://dotnet.microsoft.com/apps/aspnet/apis (visited on 09/22/2020).

[56] Google. *.NET on Google Cloud*. English. URL: https://cloud.google.com/dotnet (visited on 09/22/2020).

[57] Amazon. *Build .NET application on AWS*. English. URL: https://aws.amazon.com/developer/language/net/ (visited on 09/22/2020).

[58] Microsoft. *App Service overview*. English. July 6, 2020. URL: https://docs.microsoft.com/en-us/azure/app-service/overview (visited on 09/22/2020).

[59] Microsoft. *Choose between ASP.NET 4.x and ASP.NET Core*. English. Feb. 12, 2020. URL: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/choose-aspnet-framework?view=aspnetcore-3.1 (visited on 09/22/2020).

[60] E. Kusumawardhono. *F# High Performance*. English. 1st edition. Birmingham: Packt Publishing, Limited, 2017. ISBN: 9781786468079.

[61] Microsoft. *Create web APIs with ASP.NET Core*. English. July 20, 2020. URL: https://docs.microsoft.com/en-us/aspnet/core/web-api/?view=aspnetcore-3.1 (visited on 09/23/2020).

[62] Microsoft. *App startup in ASP.NET Core*. English. May 12, 2019. URL: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/startup?view=aspnetcore-3.1 (visited on 09/23/2020).

[63] Microsoft. *Dependency injection in ASP.NET Core*. English. July 21, 2019. URL: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.1 (visited on 09/23/2020).

[64] Microsoft. *ASP.NET MVC 4 Dependency Injection*. English. Feb. 18, 2013. URL: https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/hands-on-labs/aspnet-mvc-4-dependency-injection (visited on 09/23/2020).

[65] Microsoft. *Entity Framework Core*. English. Oct. 27, 2016. URL: https://docs.microsoft.com/en-us/ef/core/ (visited on 09/23/2020).

[66] Microsoft. *Database Providers*. English. Dec. 17, 2019. URL: https://docs.microsoft.com/en-us/ef/core/providers/?tabs=dotnet-core-cli (visited on 09/23/2020).

[67] Microsoft. *Configuring a DbContext*. English. Oct. 27, 2016. URL: https://docs.microsoft.com/en-us/ef/core/miscellaneous/configuring-dbcontext (visited on 09/23/2020).

[68] Microsoft. *ASP.NET Core web API help pages with Swagger / OpenAPI*. English. July 6, 2020. URL: https://docs.microsoft.com/en-us/aspnet/core/tutorials/web-api-help-pages-using-swagger?view=aspnetcore-3.1 (visited on 09/23/2020).

[69] Microsoft. *Tutorial: Create a web API with ASP.NET Core*. English. Aug. 13, 2020. URL: https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-3.1&tabs=visual-studio (visited on 09/24/2020).